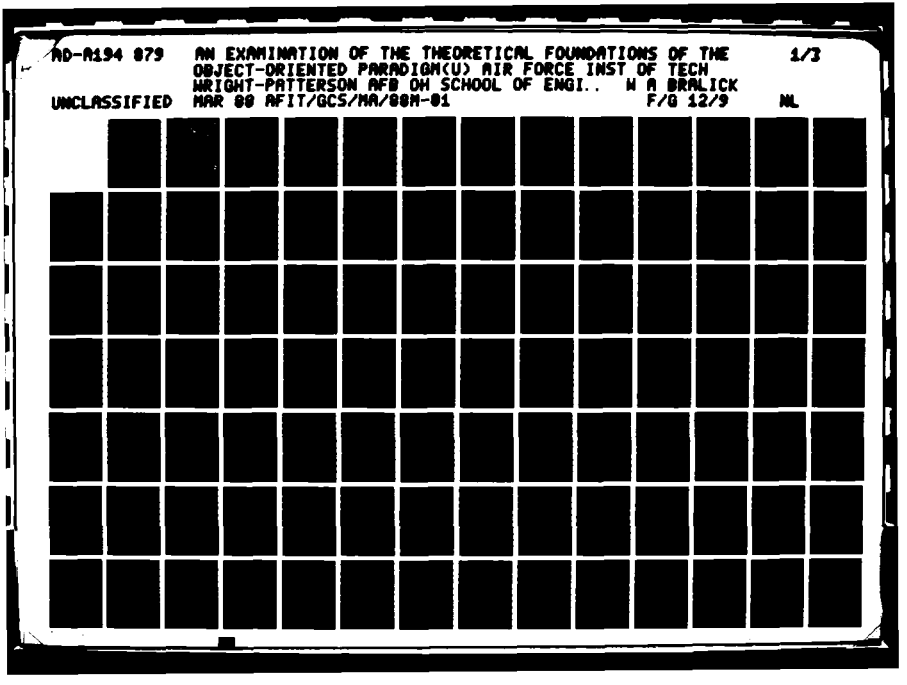
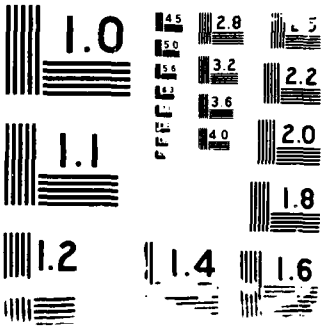

AD-A194 879
AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. W A BRALICK
1/3
UNCLASSIFIED
MAR 88 AFIT/GCS/MA/88M-01
F/G 12/9
NL


1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0
5.6

AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM

THESIS

William A. Bralick, Jr., Captain, USAF

AFIT/GCS/MA/88M-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/88M-01

AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM

THESIS

William A. Bralick, Jr., Captain, USAF

AFIT/GCS/MA/88M-01

AFIT/GCS/MA/88M-01

AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

William A. Bralick, Jr., B.S.

Captain, USAF

March 1988

Acknowledgments

First and foremost, I want to thank my thesis advisor, Dr. David Umphress, for his objectivity, incisive comments, and guidance. Without Dr. Umphress, this thesis could not have happened. I also want to express my appreciation to my other committee members: Dr. Henry Potoczny, for helping instill in me an interest in formal systems; and Dr. Panna Nagarsenker, for helping me maintain an appropriate level of rigor. Additionally, I would like to extend my gratitude to Robert Graham; our extended conversations on formal systems and related matters helped me clarify my objectives.

To my two wonderful daughters, Shannon and Jennifer, who bore with me through it all, a very special thank you. I hope that I can always be as supportive in their lives as they have been in the last year and a half of mine. My wife, Cecilia, deserves much of the credit for this thesis -- she not only helped me throughout this process, but also has made it all worthwhile. Without my family's support and encouragement, I would have been lost months ago.

Last, but not least, a word of thanks to my parents. Their concern and devotion have always been an inspiration to me, and their example strengthens my resolve to always do my best.

Table of Contents

List of Figures

List of Tables

Abstract

The object-oriented paradigm provides a natural structure for describing and decomposing systems. The objectives of this research were to: (1) provide a definition of an object model and consider its theoretical foundations; (2) implement the defined object model to empirically investigate the concept; and (3) implement a prototype environment to directly, interactively manipulate the object model.

We define an object to have a unique identity and be composed of a set of attributes, a set of behaviors, and a set of (sub)objects. We define an attribute to be composed of an identifier, a value, and a set of attributes; and we define a behavior to be an identifier, a set of attributes, and a set of behaviors. We propose and prove the theorem that the defined object model can simulate a Turing machine.

We then use the object-oriented design process to implement the defined object model under a prototype intractive environment called the HOOKE. We use the HOOKE

to help build a simulation of a Turing machine under the defined object model, including several delta functions and input tapes. Thus we validate both the defined object model and the HOOKE. We conclude that the object-oriented paradigm rests on sound theoretical ground.

1. Introduction

The object-oriented paradigm provides a natural structure for describing and decomposing systems. Unfortunately, the structure is vague. Object-oriented programming systems and languages have generally failed to state exactly what is their abstraction of an object. They all rely heavily on the intuitive, natural language notion of "object" and then proceed to implementation. My objective is to examine the object-oriented paradigm within the context of a defined object model.

1.1 Background

The software engineering process is a communicative process [Fairley, 1985], [Pressman, 1987] and as such it is fraught with ambiguity, misunderstanding, and vagueness. There are several different methods for organizing and communicating knowledge about a system which is to be developed in order to solve some problem. One such method is the object-oriented paradigm. "Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day. I am among them" [Brooks, 1987: 14].

The object-oriented paradigm is an attempt to structure or organize the solution space (computer algorithm) concepts as analogues of the problem space (real world) objects which compose the system they seek to represent. It has become more popular as its power as a representational tool has become more widely recognized. Since "representation is the essence of programming," [Brooks, 1975: 102], it is reasonable that tools which facilitate problem representation will also facilitate problem solution. In fact, Simon [1985] postulates that representation can be considered the basis of all problem solving, not just computer programming.

What then do we mean by the word "object?" In most cases, the intuitive notion of an object serves very well. In fact it is this convenience, the naturalness of the object-oriented paradigm, which has been partly responsible for the paradigm's success. Even so, an intuitive notion is not a sufficiently rigorous formulation with which to begin the process of system building. It is important to begin with a more well-defined objective.

If one's objective is to build an object-oriented programming language, one will get a programming language, albeit one with certain features. The approach to the development of most of the existing object-oriented systems has been a programming languages approach, and, not surprisingly, the result has been a spate of object-oriented programming languages. These object-oriented languages then currently embody much of the object-oriented paradigm. However, newer systems, such as HyperCard and HyperTalk [Williams, 1987: 113], are beginning to embody object-oriented principles. Unlike the Smalltalk-80 program development environment [Goldberg, 1984], these systems are separable from their implementation language.

All programming languages are equivalent in the sense that anything which can be built in one programming language can be built in any other programming language, though it may be significantly more difficult to do so. An object-oriented system does not have to be developed in an object-oriented programming language, although the use of an object-oriented programming language eases the system development. Programming languages have not proliferated in order to render that which couldn't previously be done on a computer suddenly tractable; rather, programming languages

are developed to ease the system developer's task, for example, by embedding software engineering concepts directly into a programming language -- Ada.

Finally, an environment is the context in which a user engages a system. Ideally, an environment provides convenient access to system resources by the user. Thus, direct manipulation of a resource like the defined object model is viewed as an environmental task, providing the user the capability to interactively manipulate the model.

1.2 Statement of the Problem

The goals of this thesis were: to provide a definition of an object model and consider its theoretical foundations; to implement the defined object model in order to be able to empirically investigate the concept; and to implement a prototype environment to directly, interactively manipulate the object model.

1.3 Scope

We define and examine the theoretical basis for a particular object model, which is then developed. We make

no attempt to compare existing object-oriented programming languages. Neither do we attempt to define a set of criteria or standards by which object-oriented programming languages can be measured. The language of implementation is Ada, which is not considered to be an object-oriented programming language in the sense of Smalltalk-80.

1.4 Assumptions

We assume that the prototype environment would never be a production system. Although it will be somewhat convenient and complete for its purposes, no effort will be made to fully develop the user interface.

1.5 Support Requirements

The resources required include both hardware and software. The hardware required is time-shared access to one of the Institute's VAX-11/785's (ASC) including offline (disk) storage. The software support required is access to Verdix Ada Development System (VADS) version 5.2 or later.

1.6 Overview

This chapter has provided an introduction and background information for the thesis. Chapter 2 presents a survey of the literature reviewed for this thesis. Chapter 3 describes our conceptual development, serves as a high-level description of the system to be built, and discusses the capabilities of the object model. Chapter 4 is the high-level design for the prototype environment and the object model defined in Chapter 3. Chapter 5 presents the detailed design for the prototype environment and the object model. Chapter 6 is a case study detailing the use of the prototype environment and the capabilities of the object model. Finally, Chapter 7 outlines the research conclusions and recommendations for future work.

2. LITERATURE SURVEY

The popularity and influence of the object-oriented paradigm are growing apace. The object-oriented design methodology has also recently become more popular. Many object-oriented programming languages have appeared to date; some have been accompanied by object-oriented systems or development environments. The object-oriented paradigm is a conceptual tool for managing complexity; software development environments are systemic tools dedicated to the same task. This chapter surveys the object-oriented world and looks briefly at the concept of environments.

2.1 Object-oriented Paradigm

According to The Oxford English Dictionary, an **object** is "the thing to which something is done, or upon or about which something acts or operates,"[Oxford, 1961: 14] and **oriented** means "having an emphasis, bias, or interest indicated by the" preceeding substantive (usually joined by a hyphen) or adverb." [Oxford, 1982: 112]. Thus **object-oriented** can be broadly stated as having an emphasis on, a bias toward, or an interest in those things to which something is done, or upon which something acts or operates.

In the technical literature, however, the term object-oriented is usually not defined in terms of **what it is**, but in terms of **how it is implemented** (see Cox [1985], Goldberg [1983], Cannon [1982], and MacLennan [1983]). Thus, debate rages today not only about what constitutes an object-oriented programming language, but also about what the term "object-oriented" means. The object-oriented paradigm itself, while not specifically articulated until the development of the Simula-67 programming language [Dahl and Nygaard, 1966], has been in use since the 1940s [Nygaard, 1986: 128]. During the last decade the paradigm has gained wider acceptance. This is due to two main factors. The first is the exponential growth in raw computing power which has made the computational overhead bearable. The second factor is the demonstration of practical artificial intelligence applications which has shown the usefulness of the paradigm for knowledge representation.

2.1.1 Concepts

All programming languages support the broadly stated object-oriented programming paradigm to some degree.

However, most researchers agree that, "to fully support object-oriented programming, a language must exhibit four characteristics: information hiding, data abstraction, dynamic binding, and inheritance" [Pascoe, 1986: 140]. The question then becomes how much of which of these properties is required to make a language object-oriented.

Object-orientedness is a quality, and, as such, it is always present to some degree. Given such a fuzzy concept, it is not surprising that much contradictory discussion surrounds the definition of what constitutes an object-oriented language. Indeed, some of the discussion is self-contradictory, for example:

> Simula is the first **object-oriented** language.
>
> Instances of a class are like data abstractions in having a declarative interface and a state that persists between invocations of operations, but lack the information hiding mechanism of data abstractions [Cardelli and Wegner, 1985: 480]. (emphasis added)

The above description is followed in the same article on the very next page by:

> Thus we say that a language is object oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated object type.
- Types may inherit attributes from supertypes [Cardelli and Wegner, 1985: 481] (emphasis added)

Even so, the literature is consistent in the sense that a programming language must have several properties (in some degree) to be considered "object-oriented." First, the language must support information hiding or encapsulation. Second, data abstraction facilities are necessary, though they are sometimes combined into a single requirement with information hiding as described above. Third, dynamic binding is considered to be important, or even critical capability. And, finally, objects must be able to inherit properties from other objects.

2.1.1.1 Information Hiding

The first principle of object-orientation is information hiding or encapsulation. Parnas [1972] is credited with articulating the importance of this principle in system design and decomposition. Certain features of a programming language support and enforce information hiding. For example, in Ada [DoD, 1983], separate specification and

body parts of a compilation unit provide one of the information hiding mechanisms. Types and operations which appear in the specification are exported and those which appear only in the body are not, that is, they cannot be used by other program elements [Booch, 1986]. Most modern programming languages support the principle of information hiding to some degree.

2.1.1.2 Data abstraction

Data abstraction is a special case of the general principle of abstraction which is "the intellectual tool that allows us to deal with concepts apart from particular instances of those concepts" [Fairley, 1985: 139]. In particular, "data abstraction, like procedural abstraction, enables a designer to represent a data object at varying levels of detail and, more importantly, specify a data object in the context of those operations (procedures) that can be applied to it" [Pressman, 1987].

Like information hiding, data abstraction is a software engineering principle which is enforced to some degree by the facilities provided by the language of implementation.

The importance of abstraction was described by Dijkstra [1972]; Liskov [1977], Guttag and Horning [1978], and more recently Shaw [1984] also developed theories.

2.1.1.3 Dynamic binding

In general, the key concept in dynamic binding is that the later the binding is made, the more dynamic it is.

> Without attempting to be too precise, we may speak of the **binding** of a program element to a particular characteristic or property as simply the choice of the property from a set of possible properties. The time during program formulation or processing when this choice is made is termed the **binding time** of that property for that element. [Pratt, 1975: 33]

Dynamic binding is present to some extent in all modern programming languages as access types (in Ada) or pointer variables (in Pascal, C, etc.). While this is far less powerful than the pervasive dynamic binding capabilities of Lisp [Steele, 1984] or Smalltalk [Goldberg, 1983], it does provide some dynamic binding capability while limiting the runtime overhead entailed in an interpreted language.

Since "any proposed construct whose implementation was unclear or that required excessive machine resources was

rejected" [DoD, 1983: 1.4], extensive run time or dynamic binding was intentionally left out of the Ada programming language. Even so, the availability of generics, task types, and thus pointers to task types (providing dynamic allocation of tasks) returns to the roots of object-orientation (in the sense of Simula processes) and promises to provide actor-like [Agha, 1985] object-oriented programming language technology in a general-purpose language -- Ada.

2.1.1.4 Inheritance

Less strongly required by some is the property of inheritance [Cox, 1986: 69]. Inheritance is considered to be helpful in a system since it reduces the amount of code stored in the system.

> Inheritance coupled with dynamic binding permits code to be reused. This has the attendant advantage of reducing overall code bulk and increasing programmer productivity, since you have to write less original code. Inheritance enhances code "factoring." Code factoring means that code to perform a particular task is found in only one place, and this eases the task of software maintenance [Pascoe, 1986: 142-144].

Other authors stress the importance of type inheritance as an important form of polymorphism, that is, a given call (message) induces different behaviors depending on the

receiver [Cardelli and Wegner, 1985]. Thus, object-oriented languages are viewed as necessarily polymorphic. They express the view that the "generic type polymorphism in Ada, however, is syntactic since generic instantiation is performed at compile time with actual type values that must be determinable (manifest) at compile time" [Cardelli and Wegner, 1985: 479].

We note that the traditional inheritance property can, and in many cases does conflict with the encapsulation property in popular object-oriented languages:

> Many popular object-oriented languages (e.g., Smalltalk, Flavors, and Objective-C) allow free access to inherited instance variables by descendant classes, thus denying the designer the freedom to compatibly change the representation of a class without affecting clients [Snyder, 1986: 44].

Thus inheritance is sometimes bought at the expense of information hiding and abstraction.

Finally, an important alternative to traditional inheritance is prototyping. Prototyping allows any object to serve as a template for another.

> Inheritance splits the object world into **classes**, which encode behavior shared among a group of **instances**, which represent individual memebrs of these sets. The class/instance distinction is not needed if the alternative of using **prototypes** is adopted. A prototype represents the **default** behavior for a concept, and new objects can re-use part of the knowledge stored in the prototype by saying how the new object differs from the prototype [Lieberman, 1986: 214].

Prototyping is the mechanism used for sharing behaviors and knowledge among objects in actor languages and some object-oriented systems.

2.1.2 Languages

The number of languages claiming to be object-oriented has expanded during the last several years with the growing acceptability of the object-oriented concept. Although everything from "Object Assembler" to Smalltalk-80 has been described as object-oriented [Schmucker, 1986], it is clear that some languages are more object-oriented than others. It is the paradigm itself rather than a particular implementation which is growing in acceptance the fastest.

> The author believes that object-oriented programming, as originated in SIMULA and more recently popularized by SMALLTALK and other so-called 'actor' languages, has the best potential for describing complex systems in a well-structured and natural way. Its advantages are not limited to simulation modelling [Kreutzer, 1986: 12].

As described by Berard [EVB:1985], there is a spectrum of object-oriented languages: from those which have the properties described above to a lesser degree, and, therefore, which support the ideals of object-orientation to a lesser degree, to those which do so to a greater extent. Depending on the importance that one places on the above properties, languages can be mapped against the spectrum; however, any such attempt suffers from imprecise measures of ill-defined capabilities whose relative importance to the paradigm is largely unknowable.

The complete set of programming languages can be mapped against the spectrum in Figure 1 without threatening to increase our understanding of the paradigm at all. These languages include those never designed to be object-oriented (e.g., Fortran, COBOL, etc.), extensions to existing languages (e.g., Objective C, C++, LOOPS, objective pascal, etc.), languages designed expressly to be object-oriented in nature (e.g., Smalltalk-80, CLU, ML, etc.), and languages which were designed to capture some of the capabilities in a more traditional, albeit new, programming language (e.g., Ada).

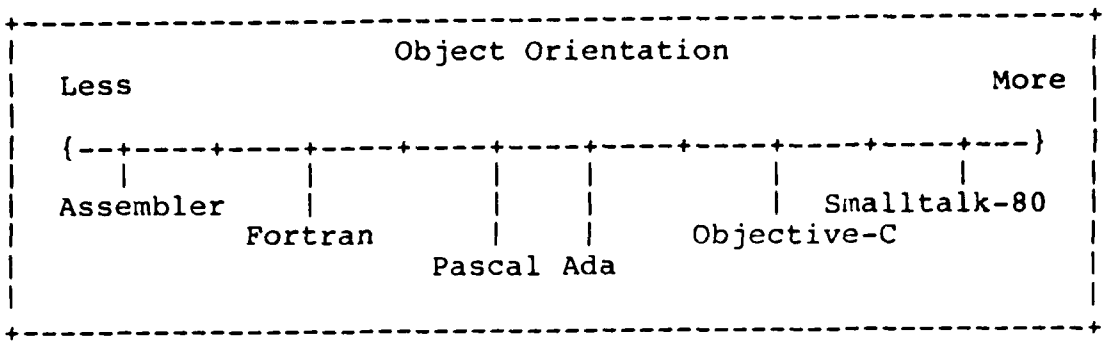


Figure 2.1. Object-Oriented Spectrum

2.1.3 Design

Since the object-oriented programming languages incorporate software engineering principles directly, it is natural to return the favor by extending object-oriented paradigm to include the design of software.

> Simply stated, **object-oriented development** is an approach to software design in which the decomposition of a system is based upon the concept of an object. An object is an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects [Booch, 1986: 211]. (emphasis added)

Object-oriented design is still a new and growing discipline. As described in Object-Oriented Design Handbook by EVB Software Engineering, Inc. [EVB, 1985], the object-oriented design/development paradigm traces its roots to Parnas [1972], Liskov and Guttag [1977], and most

especially Abbott [1983]. Also contributing to the development of the paradigm were Intel [1980], Organick [1983], and Ullman [1982].

Object-oriented design currently relies on older and more established techniques for its requirements analysis [EVB, 1983: 1-7]. It proceeds from that point by identifying the objects in the system, and recursively applying itself until fundamental objects are obtained which must be specified functionally. This procedure is described by Berard [EVB, 1985], Booch [1986:1987], and Pressman [1987].

The value of object-oriented design is said to be its ability to deal with real-time and distributed systems [Booch, 1986: 211]. Therefore, it is no wonder that it has attracted the interest of the Ada community. Still, it is important to note that an object-oriented design approach is separable from an object-oriented implementation language. While certain programming language features such as data abstraction and information hiding facilitate the direct representation of an object-oriented system, the object-oriented language features of dynamic binding and inheritance, are implementation details.

2.1.4 Systems

There appears to be little theoretical development as to what constitutes an object-oriented system per se other than the programming environment provided with a given object-oriented language, such as, for example, the Smalltalk-80 system. Since a programming language can be mapped against the object-oriented spectrum based on its inclusion of the four cardinal properties of object-orientation (data abstraction, information hiding, dynamic binding and inheritance), we can conjecture that an object-oriented system must have those same properties some degree. Therefore we can say that an object-oriented system presents to the user a view of its constituent elements which is based on information hiding and abstraction and provides the user with inheritance among and dynamic binding of those elements.

Even where an object-oriented system is claimed to exist, e.g. in Jones and Rashid [1986], it is often confused with the development method undertaken to produce it as in Meyrowitz [1986]. While an object-oriented programming language programming environment [Goldberg, 1984] is an

object-oriented system, it is so tightly coupled with the individual programming language that it renders the discussion problematic for reducing the required properties of the systems themselves. To have meaning then, the discussion of what constitutes an object-oriented system, must be separated from the degree to which the implementation language is object-oriented.

2.2 Environments

The Oxford English Dictionary defines an **environment** as "that which environs; the objects or the region surrounding anything. Especially, the conditions under which any person or thing lives or is developed." [Oxford, 1933: 231]. Thus a software engineering environment is that set of conditions under which a software programming product [Brooks, 1979] is developed.

> Although the IEEE Standard Glossary of Software Engineering Terminology [IEEE729] does not define a software engineering environment, it does provide one for a programming support environment:
>
> An integrated collection of tools accessed via a single command language to provide programming support capabilities throughout the software lifecycle. The environment typically includes tools for design, editing, compiling, loading, testing, configuration management, and project management. [Houghton and Wallace, 1987: 1]

Any program is developed under some set of conditions: technical, managerial, institutional, etc. However, implicit in the terms "software engineering environment" or "programming support environment" is the concept that the conditions under which software systems are developed can be made to facilitate the development. "The most important reason there is so much interest in software engineering environments is that there is a proven cost savings when software engineering environments are used" [Houghton and Wallace, 1987: 2].

2.3 Summary

Like so many areas of computer science, developments in the areas of environments and the object-oriented paradigm have been so rapid that no literature survey could cover the entire field. Even so, we have looked at much of the background for the object-oriented paradigm. We have also briefly looked at what constitutes an environment. While this thesis is not concerned with environments, per se, the complexity, management capability, and convenience provided by an environment is required so that we may approach the study of the object model which underlies this paradigm.

3. Concepts

The object-oriented paradigm is a powerful conceptual tool for the development of software systems. The paradigm has proven successful in object-oriented programming languages; the object-oriented design methodology is becoming more popular for the design phase. We first define the concept of an object and then explore its use across the breadth of the software development process. Finally, we address the defined object model's power as a representational tool.

3.1 Object Definition

Due to the growing popularity of the object-oriented paradigm, a number of slightly different definitions have emerged which exacerbate the conflict over what an object-oriented programming language is rather than foster an understanding of the concept. A definitional framework within which the various object-oriented languages can be placed would aid understanding. We suggest a definition which provides a framework for understanding not only the existing object-oriented programming languages, but also the object-oriented design process.

3.1.1 Object: Traditional Definition

The current definitions for an object suffer from several inadequacies. First, they are usually at too low a level of abstraction, that is, they are replete with implementation-level detail. Second, they are incomplete. And, finally, although object-oriented programming languages are much better than more traditional languages at representing the real-world, in almost all cases, they cause an unrealistic mapping of the real world to a system implemented in an object-oriented programming language [Cox, 1986]. These are not so much definitions or abstractions as they are requirements.

The notion of an object as simply "some private data and a set of operations that can access that data" [Cox, 1986: 50], or that which "consists of some local state and some behavior" [Cannon, 1982: 1], or even as "a representation of information consisting of private memory, and a set of operations to manipulate information stored in the private memory or to carry out some actions relative to that information " [Goldberg, 1984: 1], suffers from being drawn at too low a level of abstraction to serve adequately

for a high level concept. Certainly, the idea of private memory focuses more on an object's implementation than on its definition. Also, this definition provides little to distinguish an object from an abstract data type, or an abstract state machine. Thus, this type of definition has limited utility for high level conceptual manipulation.

These definitions miss entirely the natural decomposition of objects into other objects. This concept is implied in one database-oriented definition of an object as "a hierarchical cluster of tuples comprising a single root tuple that defines the object, and one or more dependent tuples that describe the object" [Dittrich and Lorie, 1985: 2]. The seemingly simple statement that "one object may be part of another object, or (in an operating system) the owner of another object" [MacLennan, 1983: 6], contains the fundamentally recursive nature of objects which transcends the hierarchical object nature implied and implemented by established object-oriented programming languages such as Smalltalk-80.

While object-oriented programming languages claim to model the real-world more closely than the prevailing

value-oriented languages, this modeling can be somewhat convoluted, since

> an object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller" [Cox, 1986: 50],

that is,

> local data structures may be manipulated only by internal methods, not from outside an object. In this way objects can be asked to perform operations that are part of their instruction set (the so-called message protocol). They are at liberty to accept or decline a task [Kreutzer, 1986: 14].

So we are left with the counter-intuitive model of, for example, a piece of sheet metal being asked by a drill press to please punch a hole in itself. The sheet metal then decides whether to honor the request. Note that whether a hole is actually made in a piece of sheet metal by a given drill press is a complex interrelationship among the material of the sheet metal, the material of the drill, the speed at with the drill bit is rotating, the force at which the drill descends onto the sheet metal, and how long the drill is applied in such a fashion to the sheet metal.

3.1.2 Object: An Alternate Definition

We shall define an object as follows:

> An object has a unique identity and is composed of a set of attributes, a set of behaviors, and a set of (sub)objects.

Here the notion of set is as understood in set theory; a set is "a collection of objects (members of the set) without repetition" [Hopcroft and Ullman, 1979: 5]. Thus an object is a set consisting of the following metaobjects: an identity, an attribute set, a behavior set, and an object set. The property which these metaobjects share in belonging to this set is that of comprising the object. The *object model* is illustrated in Figure 3.1.

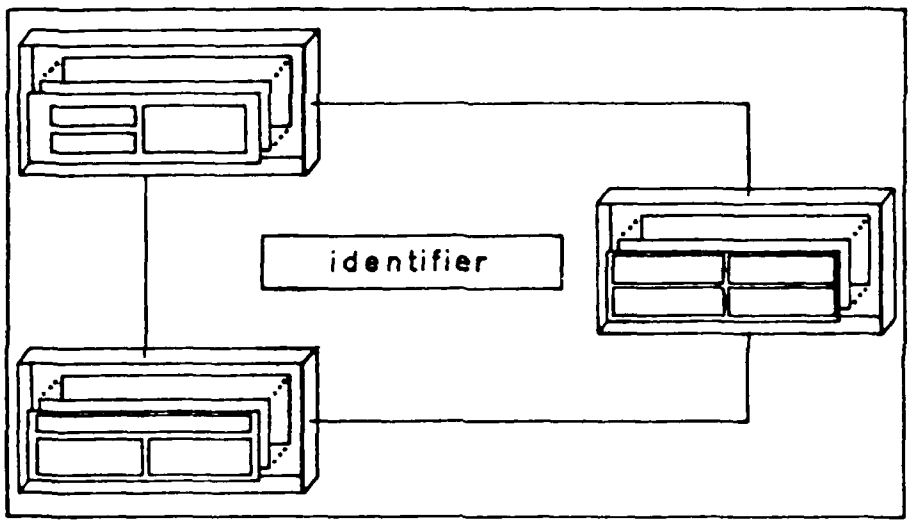


Figure 3.1. Object Model

3.1.2.1 Attributes

Attributes are properties of objects or entities, "which associate a value from a domain of values for that attribute with each entity in an entity set" [Ullman, 1980: 11]. In a grammatical sense, an attribute is "a word or phrase that is syntactically subordinate to another and serves to limit, identify, particularize, describe, or supplement the meaning of the form with which it is in construction " [Random, 1984: 88]. The concept of an attribute or property of an object is fundamental to our understanding of the world around us.

An attribute can be either a composite attribute -- representing a set of attributes (e.g. a box's size may be composed of height, width, and depth), or a primitive attribute-- providing an interpretation of, and mapping a value from, a set of possible values to the object in question (e.g. a box's size may be eight cubic feet). Note that for a primitive attribute the value alone means little, an interpretation must also be supplied.

In general, the values to which an object's attribute maps (1) depend on the interpretation to which they are subjected, and (2) are representable in many different ways. Even so, a given value can usually be interpreted as either a numeric value, a selection from a discrete set of possible values, or an arbitrary string. The numeric possibilities can include not only integer and real types, but also imaginary numbers, or whatever makes sense. The discrete set of possible values can include the normal Boolean type values {true, false} as well as what is termed an enumerated type in Ada. Finally, an arbitrary string can even map to an identifier, thus objects can be mapped to other objects by composition or attributes (e.g. father_of).

A vast number of possible sets of values and interpretations or arrangements of sets of values and interpretations can be used to describe the same underlying attribute. While this ambiguity may appear on the surface to be undesirable, note that it is exactly this type of ambiguity which is found in human communication. Thus, human communication can be directly represented in this model.

3.1.2.2 Behaviors

Behaviors are activities or operations of objects or entities. A behavior can be a "manner of behaving or acting" [Random, 1984: 122]. It can also be an activity in process or an operation: "an act or an instance, process, or manner of functioning or operating" [Random, 1984: 931]. The natural definition for the term is sufficient for our high-level concept. We will refine it later.

Thus, a behavior can be either a composite behavior -- representing a set of behaviors (e.g. filling a box may be composed of opening the box, putting items in the box, and closing the box), or a primitive behavior -- an operation which is meaningful at the current level of abstraction (e.g. filling a box is merely changing the box's full attribute from its current value to true).

Conceptually, when an object engages in a behavior, it is engaging in a process whose result is the modification of some set of attributes of some object. The philosophical question of whether an object which has been subjected to some behavior is the same object as it was prior to the application of the behavior, or is some new and different

object is not answerable here. For our purposes, we shall treat an object which has been subjected to a behavior as the same object as the pre-behavior object, since its unique identity has not changed. Similarly, the question of whether the object itself is changed or merely its attributes are changed is also beyond the scope of this thesis. We shall treat behaviors as operating on attributes, thus rendering a change in the underlying object. Our interest is in who is doing what to whom.

3.1.3 Model Viability

Can the object model as described actually be implemented? In fact, none of the capabilities implied by the object model as presented above are intrinsically difficult for most modern programming languages. In particular, the implementation of the model requires the implementation of the model's components while paying close attention to their fundamentally recursive nature. Following is a high-level description of the model's attribute, behavior and object components.

3.1.3.1 Attributes

Since each object has a unique identity, we can require each object in our system to have a unique identifier -- a special, required attribute which provides a unique way to identify or indicate a specific object. The important point is that identifiers have regular characteristics; the underlying type of the identifier attribute must be consistent across all objects. From the database arena, the concept of a key, a set of values which indicate a unique tuple, maps well to the concept of an identifier.

Most generally, a value can either be nil, meaning the attribute has no value, or it is an interpretation of an ordered sequence of bits in memory. This interpretation can proceed as for some reasonable set of predefined types in Ada. For our purposes, we will consider a legal value to be represented by any legal character string, including the null string, which shall represent the token nil.

A primitive attribute is one which is made up of a unique identifier, a legal value, and an empty set of attributes. An attribute is either a primitive attribute, or it is made up of a unique identifier, a value, and a

non-empty set of attributes. Thus, an attribute may or may not have a value at a particular level of abstraction. Whether or not the attribute has a value, it also has a possibly empty set of (sub)attributes. An attribute with a non-empty set of (sub)attributes is a composite attribute.

3.1.3.2 Behaviors

An operation is a sequence of zero or more executable statements performable on some object. That operation which is composed of zero executable statements is the universal operation nop (nil operation).

A primitive behavior is that which is made up of a unique identifier, an operation, a set of attributes, and an empty set of behaviors. A composite behavior is a behavior which has a non-empty set of component behaviors. A behavior may have no operation to perform at a particular level of abstraction and thus be composed entirely of constituent behaviors (such as a subprogram which consists entirely of function and procedure calls), or it has a simple operation and a (possibly empty) set of constituent behaviors.

3.1.3.3 Objects

A primitive object is that which is made up of a unique identifier, a set of attributes, a set of operations, and an empty set of objects. An object is either a primitive object or its constituent set of objects is non-empty. If an object's constituent set of objects is not empty there is no requirement that the object's behavior and attribute sets be empty. These behaviors and attributes are those appropriate at the object's level of abstraction and represent those aggregate attributes and behaviors of the object as a whole as distinguished from the attributes and behaviors of any constituent objects. An object which is composed of an identifier and empty sets of attributes, behaviors, and objects is a vacuous object. An object can then have any or all of its constituent collections empty.

3.2 Lifecycle Objects

There are two major categories of objects: relevant and irrelevant. The irrelevant objects are those objects which have no bearing on the solution of a problem and are therefore reduced out of the problem space. This set of objects should be maintained, though, since it is often the

case that some apparently irrelevant objects become extremely important as more is learned about the problem. There are three important subcategories of relevant objects:

-- objects of discernment which are encountered during problem definition and requirements analysis. These objects are generally extractions from the problem space and represent both tangible and intangible objects in that domain. This is the topmost level of the abstraction hierarchy.

-- objects of representation which are encountered first during high level design and subsequently in detailed design, and finally during implementation. These objects act as bridges between problem identification and the design and implementaion of a solution. These objects continue to have utility during the maintenance phase of the software.

-- objects of execution which are encountered during implementation and maintenance. These objects make up the actual software system which is delivered to the customer.

We can trace the evolution of a system from problem definition to requirements derivation and finally to the solution implementation. In defining the problem, the disparity between the anticipated, or current, and the desired object state space is stated. In requirements derivation, the set of object transformations required to resolve the problem is described. Implementation is the realization of those transformations in software. This perspective is illustrated in Figure 3.2. We enlarge our

system development perspective to view building a system as evolving a set of objects to meet an evolving set of objectives.

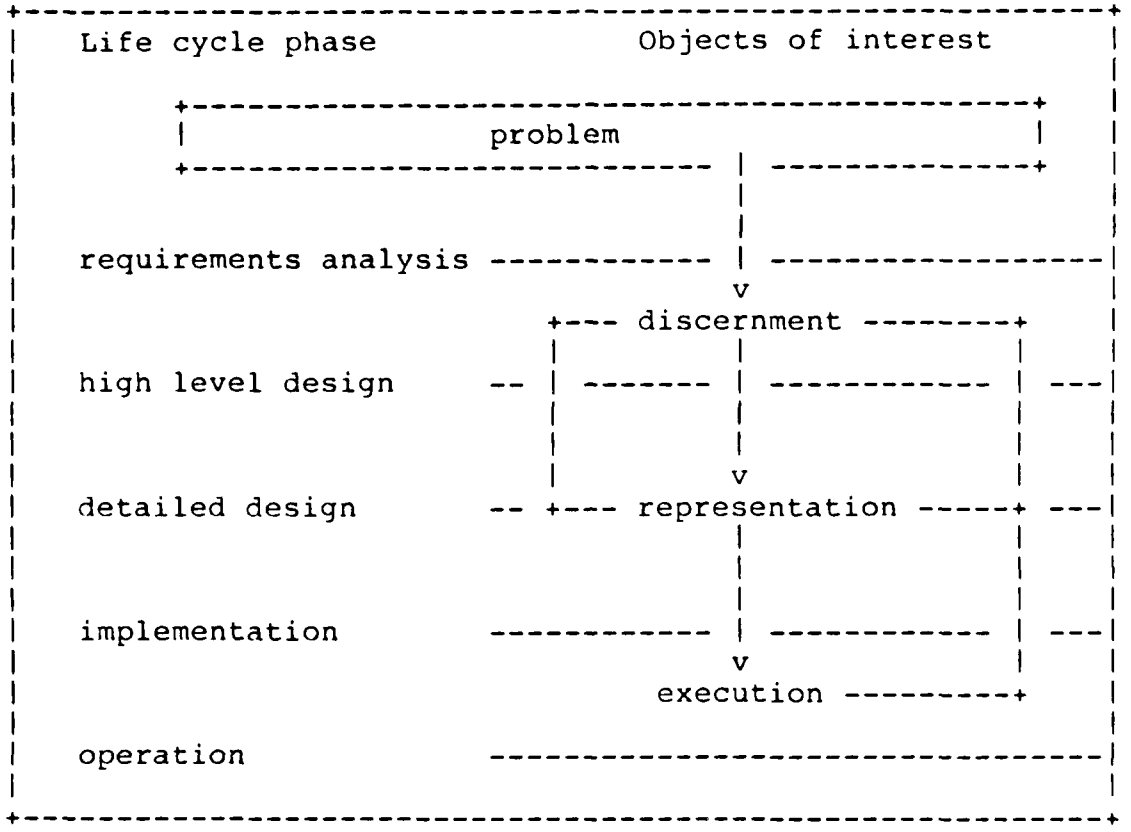


Figure 3.2. Proposed System Development Methodology

3.2.1 Objects of Discernment

One of the most serious problems in software engineering is requirements analysis. Traditionally, "requirements specify capabilities that a system must

provide in order to solve a problem. Requirements include functional requirements, performance requirements, and requirements for the hardware, firmware, software, and user interfaces" [Fairley, 1985: 33]. Requirements analysis usually begins with the assumption that the user understands his problem. This implies that if we satisfy these requirements, the problem will be solved. Unfortunately, the problem is often either incompletely or incorrectly defined initially and we therefore complete a project by merely meeting stated requirements. Thus we are often so busy trying to meet the requirements that we fail to solve the problem.

The missing stage in the current life-cycle models is that of problem analysis. The engineers leave that up to the customer and assume the problem as specified is the one needing to be solved. When the software is delivered, and the real problem has not been solved, the software is blamed. The software engineers must be able to identify the real problem, or they will never be successful at software engineering, no matter how well they can meet requirements.

The problem is usually also changing, if not in substance, then in form. In some cases, the problem is

adapting to our partial solution, e.g. enemy force structure adapts to meet our force structure adaptations undertaken to meet their force structure adaptations, etc. Thus, one reason that requirements evolve during a long development cycle is that the problem those requirements attempt to address is evolving. Typically, our response is to expand the requirements, pushing the state-of-the-art. This just increases the development time, providing more time for requirements to evolve.

In order to solve the underlying problem, we must either (1) shorten the development cycle, (2) anticipate the problem state once the development cycle is complete, or (3) stay flexible. Shortening the development cycle is possible to some extent, however, some problems are so difficult that they require a long, continuous effort to solve. The obvious difficulty with (2) is that it requires an ability to predict not only the future situation, but also the length of time required for development of a system which we cannot define. Therefore, we are left with trying to stay flexible.

The most important thing engineers can do for themselves is to anticipate the changes to the requirements

so that they are able to meet new and changing requirements quickly and effectively. This adaptability and contingency planning is as important for engineers as it is for a successful battlefield commander. Engineers who can respond quickly and effectively to changing requirements will be successful. An accurate representation of the problem is the critical element.

3.2.2 Objects of Representation

The objects of representation are design phase objects whose attributes include mappings to the objects of discernment. In this fashion, requirements can be traced to problem elements and problem elements indicate solution objects. If, during a periodic review of the problem domain, it is noted that substantial changes have occurred in some particular part of the problem domain which affects the solution being pursued at the design level, then adaptations can take place quickly.

3.2.3 Objects of Execution

Objects of execution include all deliverable software, source code, documentation, executable images, etc. These

objects map back to the design-level objects and through them to the objects of discernment. Thus an object structured design system pervades the depth of the system development process as well as the breadth of the scope of the problem and solution object state space.

3.2.4 Object Structured Design

Most of the literature on object-oriented design (OOD) is concerned with a partial lifecycle model encompassing the design, implementation, and maintenance phases. "OOD assumes that there has been some prior analysis, and that the software engineer has a basic understanding of the problem" [EVB, 1985: 1-7]. We propose a methodology called object structured design (OSD), which is a complete lifecycle model which beginning with the problem definition and ending with the retirement of the system. In fact, as the system matures in a given problem domain it should develop diagnostic and prescriptive capabilities.

We first define the problem in an object structured fashion. We specify the way things are and then specify the way we want them to be. This is a descriptive task. We describe the problem as an object state space, that is, a

set of interacting objects which have values and behaviors specified to an appropriate level of detail. Next, we describe the way we want things to be, once again, in terms of a set of interacting objects. If there is no difference between the problem object state space and the solution object state space, the problem is solved, else we list the differences.

This list of differences provides the sets of objects and their relationships which specify the relevant differences between the world as we know it and the world as we want it to be. When specified to an appropriate level of detail, this provides the problem domain analysis which should precede the OOD process.

3.3 Object Taxonomy

Given the capability for an object's behaviors to directly modify its own or other objects' attributes, objects can be classified according to their inter- and intra- object interactions. An object can affect its own or other objects' attributes and is itself affected by its own or other objects' behaviors. As shown in Table 3.1, there are eight possible object types.

Table 3.1 Object Classification

OBJECT TYPE	AFFECTS OTHERS	AFFECTS SELF	AFFECTED BY OTHERS
Static	N	N	N
Passive	N	N	Y
Small	N	Y	N
Weak	N	Y	Y
Demon	Y	N	N
Interactive	Y	N	Y
Sovereign	Y	Y	N
Complex	Y	Y	Y

Note that all of the object types are restrictions on the most general type, the complex object. Object typing is a characteristic of its abstraction, not its substance. For example, a real-time control system software module can be viewed as either directly affecting the operational control attributes of the subsystem being controlled by actively changing the subsystem's control parameters, or indirectly affecting the operational control attributes of the subsystem being controlled by sending the controlled subsystem a message requesting the parameters be adjusted. In the former case the abstraction is one of, say a sovereign-passive object pair, and in the latter case one of, say a small-small object pair. The point is that the objects assume the characteristic type relevant to the

abstraction, not a type that is somehow intrinsic to the object being represented.

3.4 Summary

We have returned to first principles and spent considerable effort defining an object and developing the concept of an object as the model which underlies the object-oriented paradigm. That being done, the question becomes "just how powerful is the defined object model?" Again, we return to the theoretical foundations of computer science and propose the following theorem:

THEOREM. The defined object model can simulate a Turing machine (Figure 3.3).

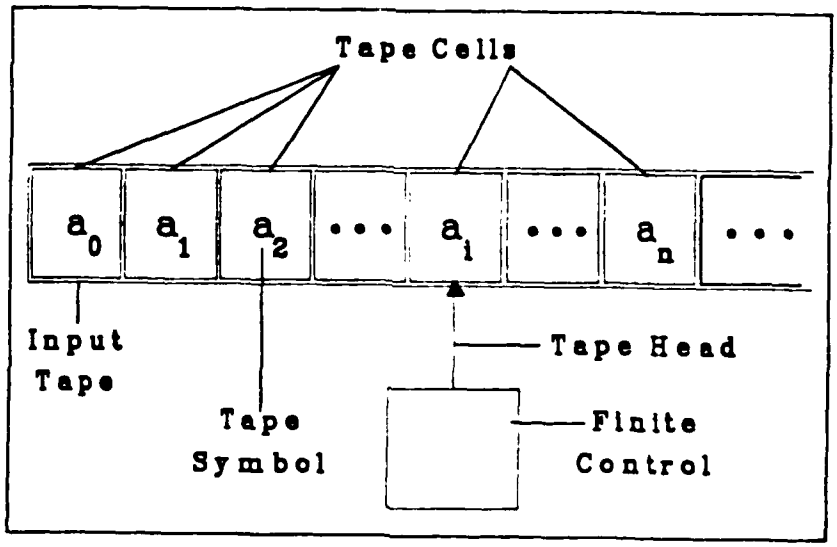


Figure 3.3. A Turing Machine

PROOF.

The proof is by construction. While there are several equally good formal definitions of a Turing machine, we will use that of Hopcroft and Ullman [1979: 148-149] as our theoretical base:

> The basic model has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time. The tape has a leftmost cell but is infinite to the right. Each cell of the tape may hold exactly one of a finite number of tape symbols. Initially, the n leftmost cells, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input tape symbols. The remaining infinity of cells each hold the blank, which is a special tape symbol that is not an input symbol.
>
> In one move the Turing machine, depending upon the symbol scanned by the tape head and the state of the finite control,
>
> 1) changes state,
>
> 2) prints a symbol on the tape cell scanned, replacing what was written there, and
>
> 3) moves it head left or right one cell.
>
> Note that the difference between a Turing machine and a two-way finite automaton lies in the former's ability to change symbols on its tape.
>
> Formally, a Turing machine (TM) is denoted
>
> M = (Q, SIGMA, GAMMA, delta, q0, B, F),
>
> where
>
> Q is the finite set of states,

GAMMA is the finite set of allowable tape symbols
B, a symbol of GAMMA, is the blank,
SIGMA, a subset of GAMMA not including B, is the set of input symbols
delta is the next move function, a mapping from Q x GAMMA to Q x GAMMA x { L, R } (delta may, however, be undefined for some arguments),
q0 in Q is the start state,
F, a subset of Q, is the set of final states.

We denote an instantaneous description (ID) of the Turing machine M by [a1 q a2]. Here q, the current state of M, is in Q; [a1 a2] is the string in GAMMA* that is the contents of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost. (Observe that the blank B may occur in [a1 a2].) We assume the Q and GAMMA are disjoint to avoid confusion. Finally, the tape head is assumed to be scanning the leftmost symbol of a2, or if a2 = epsilon, the head is scanning a blank.

We define a move of M as follows. Let [X(1) X(2) ... X(i-1) q X(i) ... X(n)] be an ID. Suppose delta(q, X(i)) = (p, Y, L), where if i-1 = n, then X(i) is taken to be B. If i=1, then there is no next ID, as the tape head is not allowed to fall off the left end of the tape. If i > 1, then we write

$$[X(1)\ X(2)\ \ldots\ X(i-1)\ q\ X(i)\ \ldots\ X(n)] \vdash_M [X(1)\ X(2)\ \ldots\ X(i-2)\ p\ X(i-1)\ Y\ X(i+1)\ \ldots\ X(n)] \quad (7.1)$$

However, if any suffix of [X(i-1) Y X(i+1) ... X(n)] is completely blank, that suffix is deleted in (7.1).

Alternatively, suppose delta(q, X(i)) = (p, Y, R). Then we write:

$$[X(1)\ X(2)\ \ldots\ X(i-1)\ q\ X(i)\ X(i+1)\ \ldots\ X(n)] \vdash_M [X(1)\ X(2)\ \ldots\ X(i-1)\ Y\ p\ X(i+1)\ \ldots\ X(n)] \quad (7.2)$$

Note that in the case i-1 = n, the string [X(i) ... X(n)] is empty, and the right side of (7.2) is longer than the left side.

If two ID's are related by $\vdash_M$, we say that the second results from the first by one move. If one ID results from another by some finite number of moves, including zero moves, they are related by the symbol $\vdash_M^*$. We drop the subscript M from $\vdash_M$ or $\vdash_M^*$ when no confusion results.

The language accepted by M, denoted by L(M), is a set of those words in SIGMA* that cause M to enter a final state when placed, justified at the left, on the tape of M, with M in state q0, and the tape head of M at the leftmost cell. Formally, the language accepted by M = (Q, SIGMA, GAMMA, delta, q0, B, F) is

{ w | w in SIGMA* and [q0 w] $\vdash^*$ [a1 p a2] for some p in F, and a1, a2 in GAMMA* }.

Given a TM recognizing a language L, we assume without loss of generality that the TM halts, i.e., has no next move, whenever the input is accepted. However, for words not accepted, it is possible that the TM will never halt.

In fact, both the structural and behavioral aspects of a TM are directly representable under the defined object model paradigm as follows:

STRUCTURE.

We assume without loss of generality that we are given the following:

Q	a finite set of states,
GAMMA	a finite set of allowable tape symbols,
B	a symbol of GAMMA, called the blank,
SIGMA	a subset of GAMMA specifically not including B, the set of input symbols,

delta a mapping from Q x GAMMA to Q x GAMMA x {L,R} which may be undefined for some (q,g) in Q x GAMMA,
q0 a state in Q which is the start state, and
F a subset of Q which is the set of final states.

Thus we have an arbitrary TM specified by some Q, some GAMMA, a blank, B, some SIGMA, some mapping, delta, a start state, q0, and some set of final states, F. Under the defined object model paradigm, we will specify the above components as the following three objects:

Q the set of states, which consists of an object called the start state which consists of an object denoted q0, an object called F, the set of final states, which consists of objects called states (denoted qi), and objects called states (denoted qj), where i, j in N and j/=0.

GAMMA the set of allowable tape symbols, which consists of an object called the blank (denoted B), an object called SIGMA, the set of input symbols, which consists of objects called tape symbols (denoted gi) where B is not in SIGMA, and objects called tape symbols (denoted gj), where i, j in N.

delta the next move (partial) function which maps an ordered pair (qi, gj) in Q x GAMMA to an ordered triple (qm, gn, d) where d ϵ { L, R }, or is undefined, that is, there does not exist (qm, gn, d) in Q x GAMMA x { L, R } such that delta(qi, gj) = (qm, gn, d).

Under the defined object model paradigm the basic model of a Turing machine can be directly represented as follows:

A Turing machine object consists of

1) a finite control object which consists of a single attribute, the current state attribute, where the current state is in Q and is initially q0 as described above.

2) an input tape object which consists of a set of tape cell objects where

a) each tape cell object has

i) a location attribute, where the location in N, and
ii) exactly one tape symbol attribute, where the tape symbol in GAMMA.

b) There exists a tape cell object whose location attribute is 0 and which is called the leftmost tape cell.

c) Initially, the n leftmost tape cell objects ($0 \leq$ location attribute $\leq n-1$) hold the input, that is, the tape symbol attribute for each of the n leftmost tape cells in SIGMA.

d) Also initially, the tape symbol attribute for each of the tape cell objects whose location attribute n is considered to be the blank, B.

3) a tape head object which consists of

a) a location attribute, where the location in N, and is initially 0, and

b) a current input attribute, where the current input in GAMMA, and is initially that tape symbol which occupies the leftmost tape cell, tape cell 0, on the input tape,

Structurally, then, a Turing machine constructed under the defined object model paradigm is isomorphic with the structural parts of the formal Turing machine definition. We now examine the behavioral aspects under the defined object model.

BEHAVIOR.

Given the Turing machine object's tape head object's current input attribute and location attribute values, which we shall call g and i respectively, and the Turing machine object's finite control object's current state attribute value which we shall call q, a Turing machine object has a move behavior which consists of

1) Changing the Turing machine object's finite control object's current state attribute to some q(q,g) in Q. Note that

   a) the new state is not necessarily different from the previous state.

   b) if q(q,g) in F the Turing machine halts and we say it "accepts."

   c) if there does not exist a q(q,g) in Q such that delta(q,g) = (q(q,g), g(q,g), d) the Turing machine halts and we say it "rejects."

2) Changing the input tape object's tape cell object's tape symbol attribute value to some g(q,g) in GAMMA. Note that the new tape symbol is not necessarily different from the previous tape symbol.

3) Changing the Turing machine object's tape head object's location attribute value from i to i+1 just in case (q,g) maps to (q(q,g), g(q,g), R) or to i-1 just in case (q,g) maps to (q(q,g), g(q,g), L). Note that

   a) if i=0 and (q,g) maps to (q(q,g), g(q,g), L) the Turing machine will halt and "reject." This is the condition where the Turing machine attempts to scan left off the end of the tape.

   b) changing the Turing machine object's tape head object's location attribute value implicitly causes the Turing machine object's tape head object's current input attribute value to be changed to the input tape object's tape cell object's tape symbol attribute value at the new location.

c) since i is in N there is no "rightmost" tape cell. If the delta function and input tape are such that the Turing machine will never halt, the model per se does nothing to disallow this.

Thus a Turing machine constructed under the defined object model paradigm is isomorphic with the behavioral elements of the formal Turing machine definition. Therefore, we see that the object-model-based Turing machine is statically isomorphic to the formally defined Turing machine both in its behavior and its structure. What about functional equivalence? To show functional equivalence we introduce the following lemma:

LEMMA.

The formally defined Turing machine and the defined object model based Turing machine are functionally equivalent.

We shall prove this lemma by induction, but first we must establish the model in more concrete terms in order to use it.

We shall represent Q, the set of states as follows:

```
[ /Q/  {} {}
  {
    [ /start/
      {}
      {}
      {
        [ /q0/ {} {} {} ]
      }
    ]

    [ /qa/ {} {} {}
    ]
    .
    .
    .

    [ /final/
      {}
      {}
      {
        [ /qb/ {} {} {}
        ]
        .
        .
        .

      }
    ]
  }
]
```

where a, b in N, the natural numbers, and a /= b.

We shall represent GAMMA the set of allowable tape symbols as follows:

```
[ /GAMMA/
  {}
  {}
  {
    [ /blank/
      {}
      {}
      {
        [ /B/ {} {} {} ]
      }
    ]
    [ /SIGMA/
      {}
      {}
      {
        [ /gc/ {} {} {}
        ]
        .
        .
        .
      }
    ]
    [ /gd/  {} {} {}
    ]
    .
    .
    .
  }
]
```

where c,d in N, the natural numbers, and c /= d.

We shall represent the delta function as follows:

```
[ /delta/ {} {}
   {
    [ /qi/        --  For a given state
      {}
      {}
      {
         [ /gj/  --  and a given tape symbol
           {
             [ /next state/      /q(qi,gj)/  {} ]
             [ /output symbol/   /g(qi,gj)/  {} ]
             [ /direction/       /d(qi,gj)/  {} ]
           }
          {}
          {}
         ]
        .
        .
        .
      }
    ]
    .
    .
    .
   }
 ]
```

where i,j in N, qi and q(qi, gj) in Q, gj and g(qi, gj) in GAMMA, and d(qi, gj) in {L,R}. Note that since the delta function is a partial function, there is not necessarily an entry for each (qi, gj); where there is no entry the (qi, gj) is considered to map to nil, that is, to be undefined.

We shall represent the Turing machine proper as follows:

```
[ /Turing machine/
  {}
  { [ /move/ {} {} ] }
  { [ /tape head/
      { [ /location/       /m/    {} ]
        [ /current input/ /gp/    {} ]
      }
      {}
      {}
    ]
    [ /finite control/
      { [ /current state/  /qn/ {} ]
      }
      {}
      {}
    ]
  }
]
```

where m, n, p in N and qn in Q and gp in GAMMA

Finally, we shall represent an arbitrary input tape as follows:

```
[ /input tape/  {} {}
  {
    [ /x/   { [ /tape symbol/ /gy/ {} ] } {} {} ]
      .
      .
      .
  }
]
```

where x, y in N and gy in GAMMA.

We shall represent an ID as follows:

```
[ /Turing machine/
  {}
  { [ /move/ {} {} ] }
  { [ /tape head/
      { [ /location/       /i/    {} ]
        [ /current input/ /Xi/    {} ]
       }
      {}
      {}
     ]
    [ /finite control/
      { [ /current state/  /q/ {} ]
       }
      {}
      {}
     ]
   }
 ]

[ /input tape/ {} {}
  {
    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /1/      { [ /tape symbol/ /X1/    {} ] } {} {} ]
     .
     .
     .
    [ /i-1/    { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]
    [ /i/      { [ /tape symbol/ /Xi/    {} ] } {} {} ]
    [ /i+1/    { [ /tape symbol/ /Xn/    {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/    {} ] } {} {} ]
  }
 ]
```

where i, n in N, q in Q, and Xi in GAMMA.

For the sake of brevity, we shall denote the above object-model-based ID as:

```
[ /input tape/
    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /1/      { [ /tape symbol/ /X1/    {} ] } {} {} ]
     .
     .
     .
    [ /i-1/    { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]

[ /Turing machine/
  { [ /tape head/
      { [ /location/       /i/    {} ]
        [ /current input/ /Xi/   {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /q/ {} ]
      }
    ]
  }
]

    [ /i/      { [ /tape symbol/ /Xi/    {} ] } {} {} ]
    [ /i+1/    { [ /tape symbol/ /Xi+1/ {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/    {} ] } {} {} ]
  }
]
```

In particular, note that that portion of the input tape denoted

```
[ /input tape/
    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /1/      { [ /tape symbol/ /X1/    {} ] } {} {} ]
     .
     .
     .
    [ /i-1/    { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]
```

is equivalent to a1 referred to in the formal definition of an instantaneous description. The portion of the input tape denoted

```
    [ /i/      { [ /tape symbol/ /Xi/    {} ] } {} {} ]
    [ /i+1/    { [ /tape symbol/ /Xi+1/ {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/    {} ] } {} {} ]
  }
]
```

is equivalent to a2 referred to in the formal definition of an instantaneous description. And, finally, that the current state of the finite control part of the Turing machine denoted

```
[ /Turing machine/
  { [ /tape head/
      { [ /location/       /i/   {} ]
        [ /current input/ /Xi/   {} ]
      }
```

```
    [ /finite control/
      { [ /current state/  /q/ {} ]
       }
     ]
   }
 ]
```

is equivalent to the q referred to in the formal definition of an instantaneous description.

We will now show the functional equivalence of a Turing machine constructed under the defined object model paradigm, and the formal definition of a Turing machine. We assume that the object model based ID

```
[ /input tape/
    [ /0/     { [ /tape symbol/ /X0/   {} ] } {} {} ]
    [ /1/     { [ /tape symbol/ /X1/   {} ] } {} {} ]
     .
     .
     .
    [ /i-1/   { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]
[ /Turing machine/
  { [ /tape head/
      { [ /location/      /i/   {} ]
        [ /current input/ /Xi/  {} ]
       }
     ]
    [ /finite control/
      { [ /current state/  /q/ {} ]
       }
     ]
   }
 ]
    [ /i/     { [ /tape symbol/ /Xi/   {} ] } {} {} ]
```

```
    [ /i+1/    { [ /tape symbol/ /Xi+1/ {} ] } {} {} ]
      .
      .
      .
    [ /n/      { [ /tape symbol/ /Xn/   {} ] } {} {} ]
   }
 ]
```

is equivalent to the formal definition ID

[X1 X2 ... X(i-1) q X(i) ... X(n)] (3.1)

just before (we shall say "at") some move j (where j in N). The Turing machine executes move j so that at move j+1 by definition one of the following six cases must obtain:

1) If i=0 and delta(q, Xi) = (p, Y, L), in the formal definition we state that there is no next ID, since the tape head is not allowed to fall off the left end of the tape.

2) If i=0 and delta(q, Xi) = (p, Y, R), in the formal definition we write the next ID as

[Y p X(i+1) ... X(n)] (3.2)

3) If i=0 and delta(q, Xi) = nil, in the formal definition we state that the machine rejects since the delta function is undefined.

4) If i>0 and delta(q, Xi) = (p, Y, L), in the formal definition we write the next ID as

[X(1) X(2) ... X(i-2) p X(i-1) Y q X(i+1) ... X(n)] (3.3)

5) If i>0 and delta(q, Xi) = (p, Y, R), in the formal definition we write the next ID as

[X(1) X(2) ... X(i-1) Y p X(i+1) ... X(n)] (3.4)

6) If i>0 and delta(q, Xi) = nil, in the formal definition, the machine rejects since the delta function is undefined.

Under the defined object model paradigm, cases three and six above are not included in the delta function object and are therefore considered to map to nil, that is, delta(qi, gj) = nil, as previously stated. The move behavior of the defined object model Turing machine (see BEHAVIOR.1.c) halts and rejects the input since there is no

next state. Case one is handled specifically by the defined object model move behavior definition (see BEHAVIOR.3.a) and, again, the machine halts and rejects the input.

Three cases remain. In case two, the tape head starts in location 0 (i=0) and delta(q, Xi) = (p, Y, L). Based on the defined object model move behavior implementation described previously, and the following ID at move j for the defined object model Turing machine

```
[ /input tape/

[ /Turing machine/
  { [ /tape head/
      { [ /location/       /0/    {} ]
        [ /current input/ /X0/    {} ]
      }
    ]
    [ /finite control/
      { [ /current state/   /q/ {} ]
      }
    ]
  }
]

    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /i+1/    { [ /tape symbol/ /Xi+1/    {} ] } {} {} ]
      .
      .
      .
    [ /n/      { [ /tape symbol/ /Xn/    {} ] } {} {} ]
  }
]
```

in a single move leads to the following ID at move j+1

```
[ /input tape/
    [ /0/     { [ /tape symbol/ /Y/        {} ] } {} {} ]

[ /Turing machine/
  { [ /tape head/
      { [ /location/      /i+1/   {} ]
        [ /current input/ /Xi+1/  {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /p/ {} ]
      }
    ]
  }
]

    [ /i+1/   { [ /tape symbol/ /Xi+1/   {} ] } {} {} ]
     .
     .
     .
    [ /n/     { [ /tape symbol/ /Xn/   {} ] } {} {} ]
  }
]
```

which we see is equivalent to equation 3.2 based on the correspondence established between equation 3.1 and the original defined object model ID.

Cases four and five are similar. In both cases the tape head starts in location i with a defined object model based ID of the form

```
[ /input tape/
    [ /0/     { [ /tape symbol/ /X0/   {} ] } {} {} ]
    [ /1/     { [ /tape symbol/ /X1/   {} ] } {} {} ]
     .
     .
     .
    [ /i-1/   { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]
```

```
[ /Turing machine/
  { [ /tape head/
      { [ /location/       /i/    {} ]
        [ /current input/ /Xi/    {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /q/ {} ]
      }
    ]
  }
]

    [ /i/      { [ /tape symbol/ /Xi/    {} ] } {} {} ]
    [ /i+1/    { [ /tape symbol/ /Xi+1/ {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/    {} ] } {} {} ]
  }
]
```

In case four, delta(q, Xi) = (p, Y, L). Based on the defined object model move behavior implementation described previously the ID at move j+1 for the defined object model Turing machine is

```
[ /input tape/
    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /1/      { [ /tape symbol/ /X1/    {} ] } {} {} ]
     .
     .
     .
    [ /i-1/    { [ /tape symbol/ /Xi-2/ {} ] } {} {} ]
```

```
[ /Turing machine/
  { [ /tape head/
      { [ /location/       /i-1/    {} ]
        [ /current input/ /Xi-1/    {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /p/ {} ]
      }
    ]
  }
]
    [ /i-1/    { [ /tape symbol/ /Xi-1/    {} ] } {} {} ]
    [ /i/      { [ /tape symbol/ /Y/       {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/      {} ] } {} {} ]
  }
]
```

which we see is equivalent to equation 3.3 based on the correspondence established between equation 3.1 and the original defined object model ID.

Finally, in case five, delta(q, Xi) = (p, Y, R). Based on the defined object model move behavior implementation described previously the ID at move j+1 for the defined object model Turing machine is

```
[ /input tape/
    [ /0/      { [ /tape symbol/ /X0/    {} ] } {} {} ]
    [ /1/      { [ /tape symbol/ /X1/    {} ] } {} {} ]
     .
     .
     .
    [ /i-1/    { [ /tape symbol/ /Xi-1/ {} ] } {} {} ]
    [ /i/      { [ /tape symbol/ /Y/     {} ] } {} {} ]
```

```
[ /Turing machine/
  { [ /tape head/
      { [ /location/       /i+1/    {} ]
        [ /current input/ /Xi+1/    {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /p/      {} ]
      }
    ]
  }
]

    [ /i+1/    { [ /tape symbol/ /Xi+1/ {} ] } {} {} ]
     .
     .
     .
    [ /n/      { [ /tape symbol/ /Xn/   {} ] } {} {} ]
  }
]
```

which we see is equivalent to equation 3.4 based on the correspondence established between equation 3.1 and the original defined object model ID. Thus, if at any move j the defined object model based Turing machine ID is equivalent to the formally defined Turing machine ID, then at the move j+1 the two IDs will still be equivalent.

An arbitrary formally defined Turing machine at move 0 (initialization) has an ID of the form

$$[q\ X0\ X1\ \ldots\ Xn] \tag{3.5}$$

The defined object model based Turing machine ID for an arbitrary Turing machine at move 0 has the form

```
[ /input tape/

[ /Turing machine/
  { [ /tape head/
      { [ /location/       /0/   {} ]
        [ /current input/ /X0/   {} ]
      }
    ]
    [ /finite control/
      { [ /current state/  /q/ {} ]
      }
    ]
  }
]

    [ /0/     { [ /tape symbol/ /X0/   {} ] } {} {} ]
    [ /1/     { [ /tape symbol/ /X1/   {} ] } {} {} ]
      .
      .
      .
    [ /n/     { [ /tape symbol/ /Xn/   {} ] } {} {} ]
  }
]
```

which we see is equivalent to equation 3.5 based on the correspondence established between equation 3.1 and the original defined object model ID.

Therefore, we have proved the lemma, since if at a given move j the defined object model based Turing machine ID is equivalent to the formally defined Turing machine ID then it will still be equivalent at move j+1, and since, in particular, the defined object model based Turing machine ID

is equivalent to the formally defined Turing machine ID at move 0, then the defined object model based Turing machine is functionally equivalent to the formally defined Turing machine. Q.E.D.

Thus, we have proved the theorem. Since the formally defined basic Turing machine is directly representable by the defined object model. Since we have shown that the formally defined Turing machine and the defined object model based Turing machine are not only structurally and behaviorally isomorphic, but also, by the lemma, functionally equivalent, then we have shown that the formally defined Turing machine and the defined object model based Turing machine are equivalent. Thus the defined object model can simulate a Turing Machine. Q.E.F.

Finally, therefore, if we assume the Church-Turing Thesis valid, we can state that the defined object model can be considered sufficiently powerful to represent any "computable function." And, therefore, the model can be considered powerful enough to represent any algorithm realizable in any programming language. This is why we state that the defined object model is a powerful conceptual tool for the development of software systems.

4. Proposed High Level Design

This chapter shall follow the object-oriented design process as outlined in EVB [1985] and Booch [1986, 1987]. The object-oriented design process consists of three steps: (1) defining the problem, (2) developing an informal strategy for the problem domain, and (3) formalizing the strategy. The third step, formalizing the strategy, is considered to be the heart of the orbject-oriented design process. It is further subdivided into four substeps: (1) identifying the objects and their attributes, (2) identifying the operations on the objects, (3) establishing the interfaces among the objects and operations, and (4) deciding on implementations of the objects and operations.

The high-level design is considered to be those steps as described above through establishing the interfaces among objects and operations. The high-level design for the defined object model is presented first (section 4.1) followed by the high-level design for the prototype environment (section 4.2). The final step, deciding on implementations of the objects and operations, is reserved for Chapter 5 -- Detailed Design.

4.1 Object Model

The implementation of the object model is straightforward. Given the data dynamic binding capabilities in Ada, the concepts discussed in Chapter 3 are directly representable.

4.1.1 Define the Problem

The realization of the object model is necessary before the object model can be used. Faithful representation of the full capabilities of the object model as described is a prerequisite to using it.

4.1.1.1 State the Problem

Design and implement the defined object model.

4.1.1.2 Analysis and Clarification of the Givens

Much of the object model's analysis and clarification information follows naturally from the conceptual development presented in Chapter 3. While no particular

effort has been made to structure what follows, effort has been made to ensure its completeness.

-- An object consists of a unique identifier, a set of attributes, a set of behaviors, and a set of objects.

-- An identifier is a sequence of characters.

-- An object's identifier provides it with its unique identity, thus within a context, it must be a unique.

-- A context is an object of reference. Thus the current context is that object to which we are currently referring.

-- An object can be stored, retrieved, and displayed individually or as part of a set of objects.

-- As a member of a given set of objects, an arbitrary object can be located in that set of objects and manipulated independently of the set.

-- A set is a collection of unique elements drawn from some universe.

-- An attribute consists of an identifier, a value, and a set of attributes.

-- An attribute can be stored, retrieved, and displayed individually or as part of a set of attributes.

-- As a member of a given set of attributes, an arbitrary attribute can be located in that set of attributes and manipulated independently of the set.

-- To be meaningful, a value may require an interpretation. An interpretation is an example of an attribute (the interpretation) of an attribute (the value).

-- A value may represent some number of type universal real or universal integer.

-- A behavior consists of an identifier, a set of attributes, and a set of behaviors.

-- A behavior can be stored, retrieved, and displayed individually or as part of a set of behaviors.

-- As a member of a given set of behaviors, an arbitrary behavior can be located in that set of behaviors and manipulated independently of the set.

-- A behavior which is mapped to a primitive operation is a primitive behavior. That which contains a non-empty set of behaviors is a composite behavior.

-- A behavior can be a composite behavior or a primitive behavior, or both.

-- A primitive operation is a compiled sequence of executable Ada source statements performable on some object.

4.1.2 Develop an Informal Strategy

An object consists of an identifier, a parental link, an attribute set, a behavior set, and an object set. An identifier is a character string. A set is a collection of zero or more unique items drawn from a universe. A parental link is a link to a like item (attribute, behavior, or object) owner of the set to which the item belongs. A behavior consists of an identifier, a parental link, an attribute set, and a behavior set. A behavior may be bound

to a primitive operation. An attribute consists of an identifier, a value, a parental link, and an attribute set. A value is a character string which may require an interpretation to be meaningful. For an arbitrary item and its set, it is necessary to be able to find the item in its set, to display the item or the set, to store the item or the set, and to retrieve the item or the set.

4.1.3 Formalize the Strategy

4.1.3.1 Identifying the Objects of Interest

4.1.3.1.1 Identifying Objects and Types

An object consists of an identifier, a parental link, an attribute set, a behavior set, and an object set. An identifier is a character string. A set is a collection of zero or more unique items drawn from a universe. A parental link is a link to a like item (attribute, behavior, or object) owner of the set to which the item belongs. A behavior consists of an identifier, a parental link, an attribute set, and a behavior set. A behavior may be bound to a primitive operation. An attribute consists of an identifier, a value, a parental link, and an attribute set.

A value is a character string which may require an interpretation to be meaningful. For an arbitrary item and its set, it is necessary to be able to find the item in its set, to display the item or the set, to store the item or the set, and to retrieve the item or the set.

Table 4.1 shows the objects and types that comprise the defined object model.

Table 4.1 Object Model Table of Objects and Types

OBJECT	SPACE	IDENTIFIER
object	solution	Object
identifier	solution	Identifier
link	solution	Parent
set	solution	List
string	solution	Character_String
collection	problem	
item	problem	
universe	problem	
attribute	solution	Attribute
behavior	solution	Behavior
owner	problem	
operation	problem	
value	solution	Value
interpretation	problem	

4.1.3.1.2 Associating Attributes with the Objects and Types of Interest

OBJECT

-- An object is uniquely identifiable in a given context.

-- A context is a reference to an (owning) object.

-- An object can own an attribute set. This set of attributes should properly be attributes of the owning object.

-- An object can own a behavior set. This set of behaviors should properly be behaviors in which the owning object engages.

-- An object can own an object set. This set of objects should properly be (sub)objects from which the owning object is built.

IDENTIFIER

-- An identifier provides an object with its unique identity in a given context, thus it must be a unique sequence of characters.

LINK

-- A parental link is a link back from an item to the owner of the like item owner of its set. If the owner of the set is a non-like item (e.g. an object owner for a set of behaviors or attributes) then the link is null. Otherwise, the link unambiguously associates the child item with the parent.

SET

-- A set is a collection of unique items drawn from a collection of objects called the universe.

-- Where the term set is used in the informal strategy, it refers to a polylithic unordered structure of unique elements.

STRING

-- The character strings referred to in the informal strategy are variable length graphic character strings.

-- A character string can be stored, displayed, and retrieved.

ATTRIBUTE

-- An attribute can own an attribute set. While there is no systemic restriction on this set, it is assumed that its use will be consistent with the owning attribute, for example a "volume" attribute could have a set of (sub)attributes including "height", "width", and "depth". To be consistent, the product of the values of the "height", "width", and "depth" attributes should be equal to the value of the owning "volume" attribute. Unfortunately, the "temperature" attribute could also be (incorrectly) included.

BEHAVIOR

-- Can be bound dynamically to a primitive operation.

-- A behavior can own a behavior set. A set of behaviors which underlie an owning behavior should cause the owning behavior's effects though operating at a lower level of abstraction.

-- A behavior can own an attribute set. This attribute set should be used to provide control parameters to the behavior's bound primitive operation.

VALUE

-- A value is a character string. A character string here includes the null string (a string with no characters).

-- To be "meaningful" a value may require an interpretation, which is a set of attributes which conveys information through the attributes' values. For example a "weight" attribute may have a value of "10" and a set of (sub)attributes which includes the attribute "units" which has a value of "pounds".

4.1.3.2 Identify Operations of Interest

4.1.3.2.1 Identify Operations

An object consists of an identifier, a parental link, an attribute set, a behavior set, and an object set. An identifier is a character string. A set is a collection of zero or more unique items drawn from a universe. A parental link is a link to a like item (attribute, behavior, or object) owner of the set to which the item belongs. A behavior consists of an identifier, a parental link, an attribute set, and a behavior set. A behavior may be bound to a primitive operation. An attribute consists of an identifier, a value, a parental link, and an attribute set. A value is a character string which may require an interpretation to be meaningful. For an arbitrary item and its set, it is necessary to be able to find the item in its set, to display the item or the set, to store the item or the set, and to retrieve the item or the set.

Table 4.2 lists the operations and their corresponding objects and identifiers.

Table 4.2 Object Model Table of Operations

OPERATION	SPACE	OBJECT	IDENTIFIER
consists	problem		
bound	problem		
find	solution	Attribute	Find_Attribute
	solution	Behavior	Find_Behavior
	solution	Object	Find_Object
store	solution	Attribute	Store_Attribute
	solution	Attribute Set	Store_Attribute_List
	solution	Behavior	Store_Behavior
	solution	Behavior Set	Store_Behavior_List
	solution	Object	Store_Object
	solution	Object Set	Store_Object_List
display	solution	Attribute	Display_Attribute
	solution	Attribute Set	Display_Attribute_List
	solution	Behavior	Display_Behavior
	solution	Behavior Set	Display_Behavior_List
	solution	Object	Display_Object
	solution	Object Set	Display_Object_List
retrieve	solution	Attribute	Retrieve_Attribute
	solution	Attribute Set	Retrieve_Attribute_List
	solution	Behavior	Retrieve_Behavior
	solution	Behavior Set	Retrieve_Behavior_List
	solution	Object	Retrieve_Object
	solution	Object Set	Retrieve_Object_List

4.1.3.2.2 Associating Attributes With the Operations of Interest

FIND

-- Find an attribute in an attribute set returns a pointer to the found attribute.

-- Find a behavior in a behavior set returns a pointer to the found attribute.

-- Find an object in an object set returns a pointer to the found object.

STORE

-- In all cases, store uses Text_IO to place a copy of the item or item set into the native file system.

DISPLAY

-- Display uses Text_IO to place a copy of the item or item set on the Text_IO.STANDARD_OUTPUT device.

RETRIEVE

-- Retrieve uses Text_IO to fetch a copy of the item or item set from the native file system.

4.1.3.2.3 Grouping Operations, Objects, and Types

OBJECT	find display store retrieve
IDENTIFIER	display store retrieve
CHARACTER STRING	
PARENT	
ATTRIBUTE SET	display store retrieve
ATTRIBUTE	
BEHAVIOR SET	display store retrieve
BEHAVIOR	
OBJECT SET	display store retrieve
OBJECT	

BEHAVIOR	find display store retrieve
IDENTIFIER	display store retrieve
CHARACTER STRING	
PARENT	
ATTRIBUTE SET	display store retrieve
ATTRIBUTE	
BEHAVIOR SET	display store retrieve
BEHAVIOR	
ATTRIBUTE	find display store retrieve
IDENTIFIER	display store retrieve
CHARACTER STRING	
VALUE	display store retrieve
CHARACTER STRING	
PARENT	
ATTRIBUTE SET	display store retrieve
ATTRIBUTE	

4.1.3.3 Defining the Interfaces

4.1.3.3.1 Formal Description of the Visible Interfaces

```
package   Attribute_Package

     type ATTRIBUTE_RECORD
     type ATTRIBUTE

     package   Attribute_List            --  reusable

     function  Find_Attribute
     procedure Store_Attribute_List
     procedure Store_Attribute
     procedure Display_Attribute_List
     procedure Display_Attribute
     procedure Retrieve_Attribute_List
     procedure Retrieve_Attribute

package   Behavior_Package

     type BEHAVIOR_RECORD
     type BEHAVIOR

     package   Behavior_List             --  reusable

     function  Find_Behavior
     procedure Store_Behavior_List
     procedure Store_Behavior
     procedure Display_Behavior_List
     procedure Display_Behavior
     procedure Retrieve_Behavior_List
     procedure Retrieve_Behavior

package   Object_Package

     type OBJECT_RECORD
     type OBJECT

     package   Object_List               --  reusable

     function  Find_Object
     procedure Store_Object_List
     procedure Store_Object
     procedure Display_Object_List
     procedure Display_Object
     procedure Retrieve_Object_List
     procedure Retrieve_Object
```

```
package   Utilities

     package   Character_String          --  reusable

     function  Hash
     procedure Store_String
     procedure Display_String
     function  Retrieve_String
     procedure Scan_Past_Next
```

4.2 Hierarchical Object-Oriented Kernel Environment (HOOKE)

The HOOKE is intended to provide direct, interactive manipulation of the defined object model of section 4.1. The HOOKE is a prototype environment and is required to be written entirely in Ada to meet Stoneman [DoD, 1980] implementation language requirements.

4.2.1 Define the Problem

The kernel environment is tightly bound to the structure of the object model for which it is built to provide access. While this tight structural coupling is not to be recommended in a production system, it can hardly be avoided in a system whose intent is to investigate the properties of the defined object model.

4.2.1.1 State the Problem

Design and implement the Hierarchical Object-Oriented Kernel Environment (HOOKE).

4.2.1.2 Analysis and clarification of the givens.

Knowing what the defined object model is supposed to be conceptually, and knowing at least the high-level design concepts for the implementation of the model, we can specify what kind of environment provides direct interactive access to the defined object model. Although the HOOKE is a prototype environment, it should still provide all the necessary capabilities for using the object model.

-- The interactive environment is assumed to be a menu-driven environment provided the user on a VT-100 terminal. The I/O capabilities to/from the VT-100 terminal, include reading input from the keyboard, and writing output to an arbitrary location on the screen.

-- If context sensitive help has been developed for a given context, it should be accessible to the user.

-- The basic set of tools provided the user includes the capabilities to create, fetch, modify, destroy, examine, and store defined object model objects.

-- Additionally, the environment must provide the user the capabilities of changing context and stimulating an object's behaviors.

-- Creating an object means specifying a unique identifier, associating the identifier with empty sets of attributes, behaviors, and objects, and adding the new object to the current context's set of objects.

-- To fetch a previously stored object means to recover an object, including all of its components, from the native file system.

-- Modifying an object means that, given an unambiguously specified object, the user can change the object's set of attributes and set of behaviors. Modifying an object includes:

  o Adding a new attribute or behavior to an object's set of attributes or set of behaviors.

  o Deleting an existing attribute or behavior from an object's set of attributes or set of behaviors.

  o Changing an attribute or behavior includes specifying a new value for an existing or newly created attribute, and binding a new behavior to a primitive operation, or unbinding an existing behavior from a primitive operation.

  o Finally, note that the capability to modify an object's set of objects is provided by the create, destroy, and change context capabilities. Since modifying an object's identifier fundamentally alters the object, it is more consistent with our abstraction to require the user to clone the object, that is to create a new object with the desired name and copy the original object into it.

-- Destroying an unambiguously specified object requires that it be removed from its context's set of objects. It is understood that when the object is destroyed, it takes its sets of attributes and

behaviors with it. Destroying an object should not result in system resources becoming permanently unavailable.

-- Examining an object displays the object's components on the user's terminal. It should be possible to specify what level of detail is desired.

-- Storing an object for later reuse records the object, including all of its components, on the native file system. The format used must be provided the fetch capability.

-- Changing the current context means either returning to a previously selected context, or selecting an object from the current context's set of objects under which to interpret further user commands.

-- Stimulating an object's behavior is equivalent to executing (1) the underlying user-defined primitive operation (if any) against a specified object, and (2) the underlying primitive operations of the set of behaviors which underlies the given behavior.

-- A user-defined, primitive operation is a user-written Ada procedure which takes and returns an OBJECT as a single (in out) parameter.

-- Otherwise, the term manipulating an object is a generic term used to leave open the possibility that other primitive HOOKE capabilities might be implemented.

-- The outermost level of the HOOKE environment greets the user and outputs a menu screen for a vacuous object whose identifier is "HOOKE".

4.2.2 Develop an Informal Strategy

The HOOKE maintains the current context and presents a menu which displays the current context identifier, the current context object's set of attributes, behaviors, and objects, and the list of available options. The HOOKE responds to the selection of one of the available options which include the capability to create, fetch, modify, destroy, examine, and store an object. The HOOKE provides the capability to execute a given behavior from the current context's set of behaviors. It provides the capability to change the current context to that of one of the objects in the set of objects by utilizing the object, or to that of the current context's parent object by quitting the current context. User-defined primitive operations are bound to behavior names through an operation map which must be initialized to recognize the user-defined primitive operations at the start of the program execution. Initially, the system generates a vacuous object whose identifier is initialized to "HOOKE" to which the current context is then set.

4.2.3 Formalize the Strategy

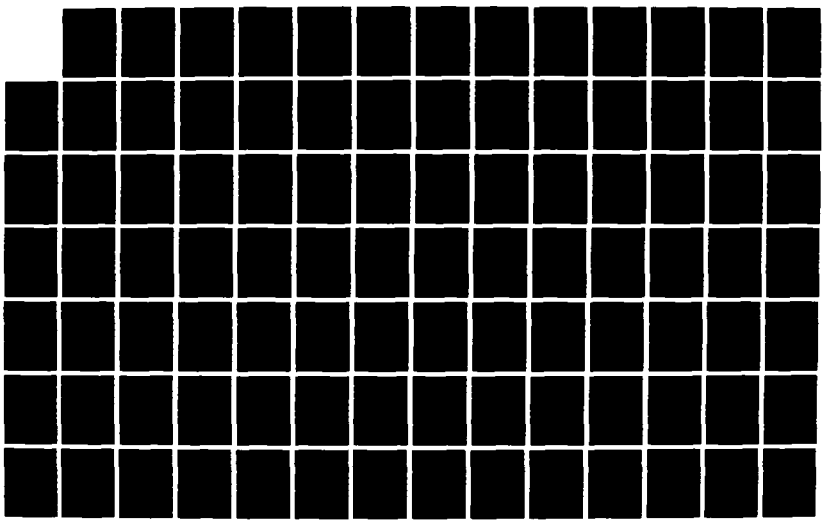
4.2.3.1 Identifying the Objects of Interest

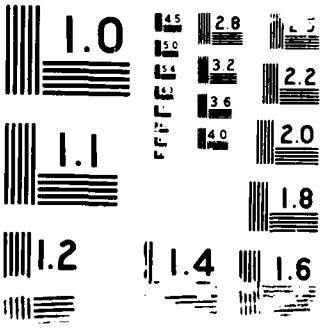
4.2.3.1.1 Identifying Objects and Types

The HOOKE maintains the current context. The HOOKE presents a menu which displays the current context identifier, the current context object's set of attributes, behaviors, and objects, and the list of available options. The HOOKE responds to the selection of one of the available options which include the capability to create, fetch, modify, destroy, examine, and store an object. The HOOKE provides the capability to execute a given behavior from the current context's set of behaviors. The HOOKE provides the capability to change the current context. The current context can be changed to that of one of the objects in the set of objects by utilizing the object, or to that of the current context's parent object by quitting the current context. User-defined primitive operations are bound to behavior names through an operation map which must be initialized to recognize the user-defined primitive operations at the start of the program execution. Initially, the system generates a vacuous object whose identifier is initialized to "HOOKE" to which the current context is then set.

AD-A194 879 AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. W A BRALICK 2/3

UNCLASSIFIED MAR 88 AFIT/GCS/MA/88M-01 F/G 12/9 NL




1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0

The HOOKE objects and types are shown in Table 4.3.

Table 4.3 HOOKE Table of Objects and Types

OBJECT	SPACE	IDENTIFIER
HOOKE	problem	
the current context	solution	The_Current_Context
menu	problem	
the current context identifier	solution	Identifier
the current context object's set of attributes	solution	List_of_Attributes
behaviors	solution	List_of_Behaviors
objects	solution	List_of_Objects
the list of available options	solution	HOOKE_Operations
the selection of one of the available options	solution	The_User_Selection
the capability	problem	
an object	problem	
a given behavior	problem	
one of the objects in the set of objects	problem	
the current context's parent object	problem	
user-defined primitive operations	problem	
behavior names	problem	
operation map	solution	The_Operation_Map
the start of program execution	problem	
the system	problem	
a vacuous object	solution	
identifier	solution	Identifier
"HOOKE"	solution	(=Identifier)

4.2.3.1.2 Associating Attributes with the Objects and Types of Interest

```
The_Current_Context

--   The current context is a defined object model object.

--   The current context is the reference point from which
     attributes, behaviors, and objects are understood.

--   The "HOOKE" object has a null parent.

Identifier

--   An identifier is a character string.  In a given
     context, it is unique.

List_of_Attributes

--   Attribute_Package.Attribute_List

List_of_Behaviors

--   Behavior_Package.Behavior_List

List_of_Objects

--   Object_Package.Object_List

HOOKE_Operations

--   An enumerated type reflecting the capabilities required
     of the HOOKE.

The_User_Selection

--   An object of type HOOKE_Operation - identifies the
     action the user wishes to take in a given context.

The_Operation_Map

--   An object of type Map.MAP_TYPE - provides the linkage
     from the behaviors maintained in the lists of behaviors
     to the user defined primitive operations.

--   Domain: Character_String - BEHAVIOR.Identifier
```

-- Range: Operation_Type - uniquely specifies each of the user-defined primitive operations.

4.2.3.2 Identify Operations of Interest

4.2.3.2.1 Identify Operations

The HOOKE maintains the current context. The HOOKE presents a menu which displays the current context identifier, the current context object's set of attributes, behaviors, and objects, and the list of available options. The HOOKE responds to the selection of one of the available options which include the capability to create, fetch, modify, destroy, examine, and store an object. The HOOKE provides the capability to execute a given behavior from the current context's set of behaviors. The HOOKE provides the capability to change the current context. The current context can be changed to that of one of the objects in the set of objects by utilizing the object, or to that of the current context's parent object by quitting the current context. User-defined primitive operations are bound to behavior names through an operation map which must be initialized to recognize the user-defined primitive operations at the start of the program execution. Initially, the system generates a vacuous object whose identifier is initialized to "HOOKE" to which the current context is then set.

The operations of the HOOKE are shown in Table 4.4.

Table 4.4 HOOKE Table of Operations

OPERATION	SPACE	OBJECT	IDENTIFIER
maintains	problem		
presents	solution	The_Current_Context	Menu
displays	problem		
responds	solution	The_User_Selection	Menu
include	problem		
create	solution	The_Current_Context OBJECT	Create
fetch	solution	The_Current_Context OBJECT	Fetch
modify	solution	The_Current_Context OBJECT	Modify
destroy	solution	The_Current_Context OBJECT	Destroy
examine	solution	The_Current_Context OBJECT	Examine
store	solution	The_Current_Context OBJECT	Store
provides	problem		
execute	solution	The_Current_Context.Behavior OBJECT	Execute
change	problem		
utilize	solution	The_Current_Context OBJECT	Utilize
quit	solution	The_Current_Context	Quit
bound	solution	Behavior.Identifier	Operations_Map.Bind
initialize	solution	The_Operations_Map	Initialize_Operations_Map
recognize	problem		
generates	problem		
set	problem		
consists	problem		
bound	problem		

4.2.3.2.2 Associating Attributes With the Operations of Interest

Menu

-- A menu is displayed for the current context. The current context identifier, list of attributes, list of behaviors, and list of objects are all displayed.

-- The set of available operations are displayed from which the user can select.

-- The menu operation returns the user-selected HOOKE operation option.

Create

-- Create an object and add it to the current context's list of objects. The object identifier must be unique within the list of objects.

Fetch

-- Fetch an object from the native file system and add it to the current context's list of objects. The object identifier must be unique within the list of objects.

-- The object's filename is created by appending ".OBJ" to the object identifier.

Modify

-- Modifying an object entails modifying the set of attributes or the set of behaviors in the current context. To modify an object, change the context to that object, then select modify.

-- Modifying the current context's attributes consists of adding an attribute to the list, deleting an attribute from the list, changing the value of one of the attributes in the list, or modifying the attribute's list of attributes.

-- Modifying the current context's behaviors consists of adding (deleting) a behavior to (from) the list, or modifying the behavior's list of attributes or behaviors. Adding a behavior to the list of behaviors may entail binding the behavior to a user-defined

primitive operation. Deleting a behavior may entail unbinding the behavior from a user-defined primitive operation.

Destroy

-- To destroy an object, it is sufficient to remove it from the current context's list of objects, dereferencing it.

Examine

-- To examine an object, the object and its component parts must be displayed on the user's terminal.

-- If the word "self" is specified when the user is prompted for the name of an object to examine, the current context and its component parts will be displayed.

Store

-- To store an object, the object and its component parts must be written out to the native file system.

-- The object's filename will be built just as for fetch.

Execute

-- Execute a behavior from current context list of behaviors

-- Supply name of object (self)

Utilize

-- Change the current context to be one of the objects in the current context's list of objects.

Quit

-- Change context to be the parent object. If the current context is "HOOKE", the effect is to end program execution.

-- This is the inverse of the utilize operation.

Operations_Map.Bind

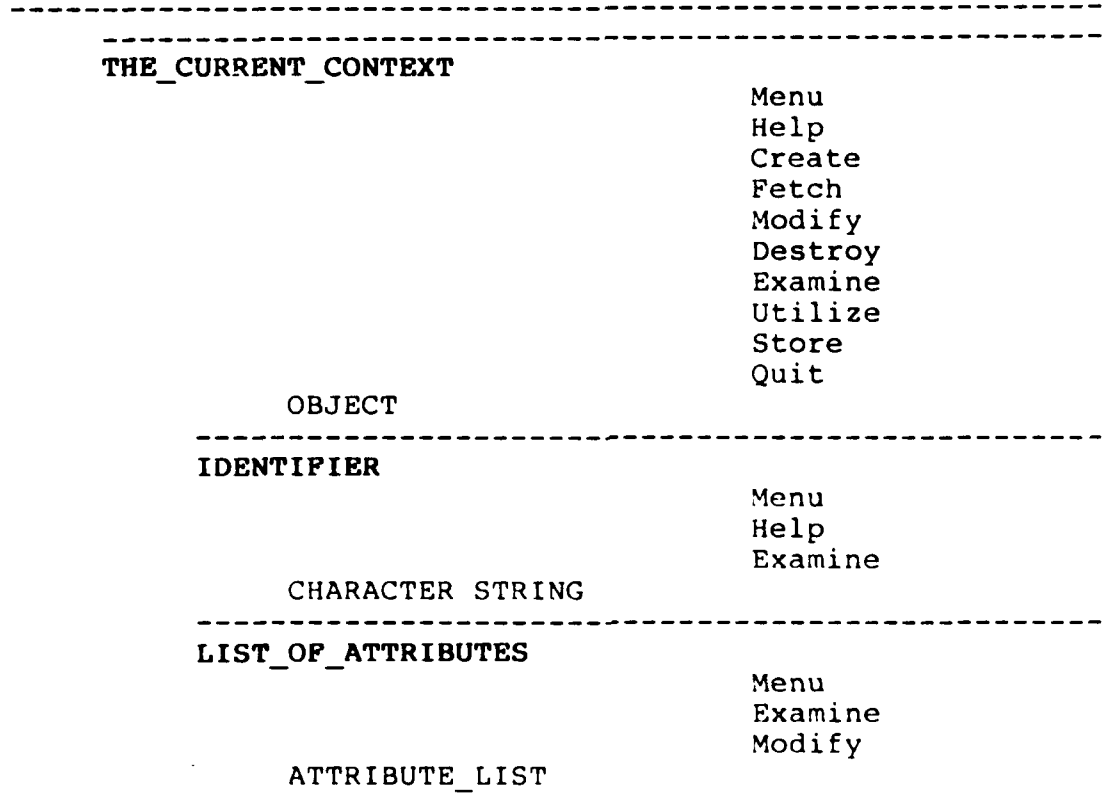
-- Used to associate a character string with a user-defined primitive operation in the operation map.

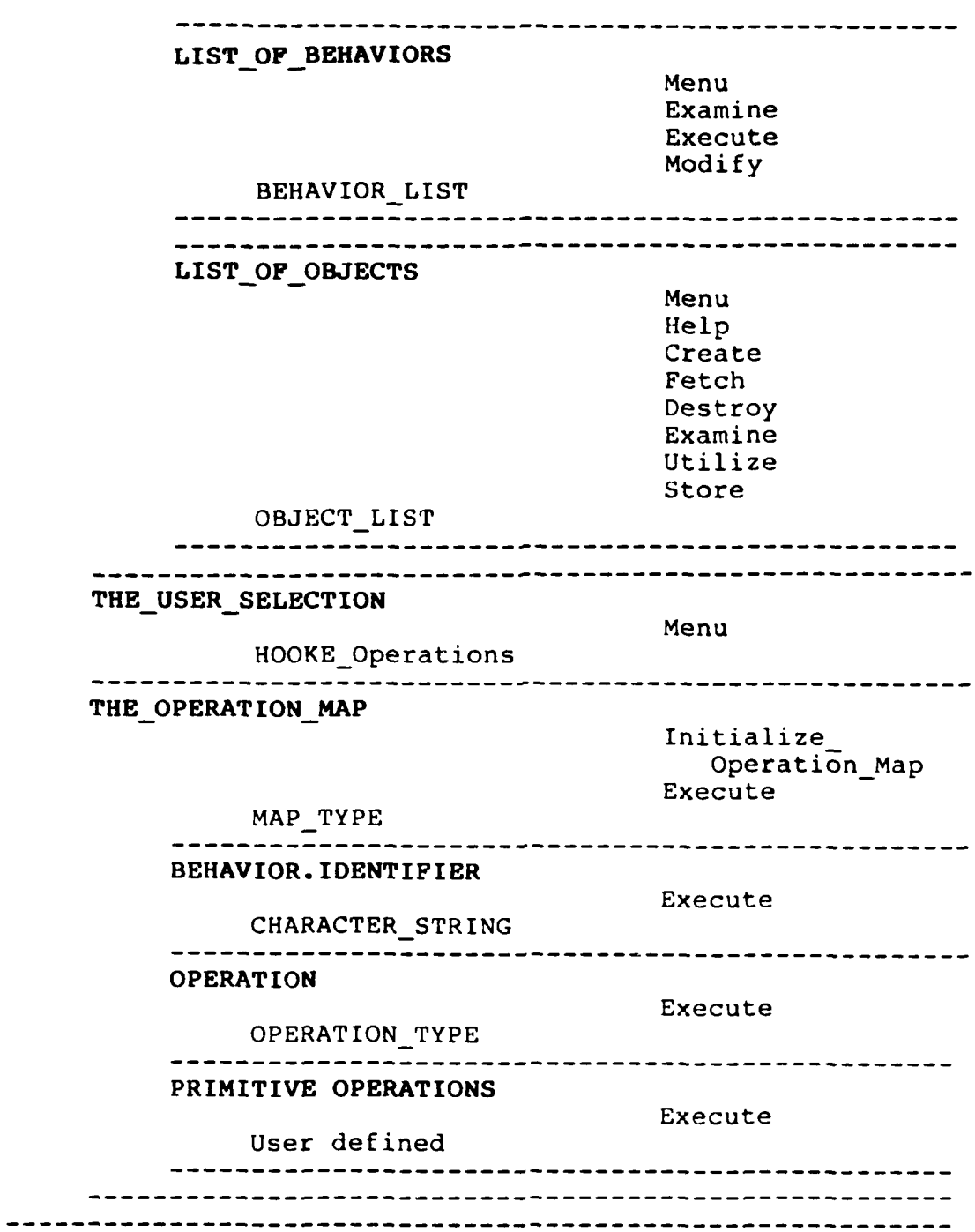
-- When a behavior is deleted it may need to be unbound from the map.

Initialize_Operations_Map

-- Binds all user-defined primitive operations in the operation map to at least one character string. Any user-defined primitive operation not initially bound to at least one character string will not be reachable until the system is recompiled.

4.2.3.2.3 Grouping Operations, Objects, and Types




LIST_OF_BEHAVIORS
Menu
Examine
Execute
Modify
BEHAVIOR_LIST
LIST_OF_OBJECTS
Menu
Help
Create
Fetch
Destroy
Examine
Utilize
Store
OBJECT_LIST
THE_USER_SELECTION
Menu
HOOKE_Operations
THE_OPERATION_MAP
Initialize_
Operation_Map
Execute
MAP_TYPE
BEHAVIOR.IDENTIFIER
Execute
CHARACTER_STRING
OPERATION
Execute
OPERATION_TYPE
PRIMITIVE OPERATIONS
Execute
User defined

4.2.3.3 Defining the Interfaces

4.2.3.3.1 Formal Description of the Visible Interfaces

```
package   HOOKE_Package

     type      HOOKE_Operations

     package   HOOKE_IO        --  Enumeration_IO

     procedure Menu
     procedure Help
     procedure Create
     procedure Fetch
     procedure Modify
     procedure Destroy
     procedure Examine
     procedure Store


package   Operation_Package


     type      OPERATION_TYPE

     type      Operation_Map   --  Reusable

     The_Operation_Map

     type      OPERATION_TASK_TYPE
     type      OPERATION_ACCESS_TYPE

     procedure Initialize_Operation_Map
     procedure Execute
     procedure Utilize

procedure MAIN     which is the highest level
                    program unit.
```

5. Detailed Design

The detailed design of the system's modules follow. The actual pseudocode for each of the modules is contained in Appendix A. The modules are arranged in the same order as they are referenced in the high-level design of Chapter 4, with the exception of section 5.1 which did not appear explicitly in the high-level design. The detailed pseudocode for each of the following modules is presented in Appendix A. The following "wiring diagram" of the system program unit dependencies in Figure 5.1 demonstrates a rather flat overall structure. This is reasonable, though, due to the exploratory nature of the research.

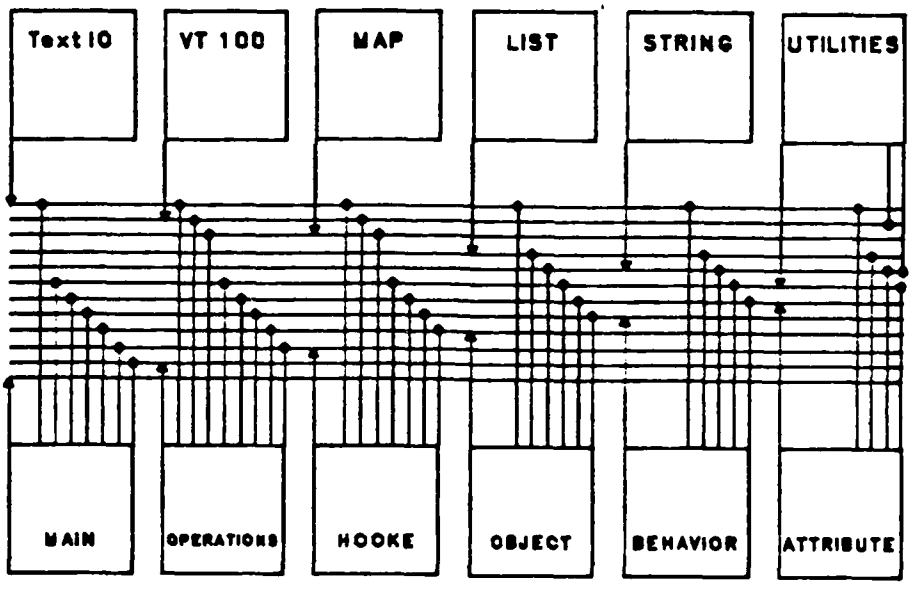


Figure 5.1. Program Unit Dependencies.

An effort was made to use reusable software components. Two sources provided those components. The first source was the book Software Components with Ada by Grady Booch [1987]. The second source was the Simtel20 software repository. It is important to note that in neither case were the components usable in the condition in which they were found, but had to be adapted to my needs. The problems varied from simple errors to complex abstraction misalignments. Even so, there is no possible way this research could have been completed as expeditiously as it has been without the use of reusable or other-source components. Even with the adaptations which were required, the advantage of having a reusable, source-code transportable library of data structure components was an invaluable aid.

5.1 Components

The components addressed in this section together with their descriptive taxonomy are all derived from Booch [1987].

5.1.1 String

According to Booch, a string is "a sequence of zero or more items; the item type is immaterial to the behavior of the string" [Booch, 1987: 105]. The exact form of string used in this thesis is the sequential, unbounded, unmanaged, noniterator string as implemented in Booch [1987: 109-128]. By using this component I hoped to achieve some generality in that in those places where a string is used, I am not in the future restricted to using only character strings, but could in fact use any enumerated or scalar type.

The string component did not require any modification. The only adaptation which I made was to add the constructor **ASSIGN** to the component. This constructor is nothing more than a string-to-string copy as in the overloaded **COPY** constructor. The reason for adding **ASSIGN** is to provide a single, non-overloaded constructor equivalent to the ":=" operator in non-limited types. While this is not necessary for the proper functioning of the string component, it was an adaptation which helped keep the abstraction understandable in its use in the Map component (section 5.1.3). Figure 5.2 presents a summary of the string component modifications.

5.1.2 List

According to Booch's abstraction, a list is "a sequence of zero or more items in which items can be added and removed from any position such that a strict linear ordering is maintained" [Booch, 1987: 71]. The difference between a

```
String_Sequential_Unbounded_Unmanaged_Noniterator

ADD:

   procedure  Assign

      parameters:

         The_String     :  in      STRING_TYPE
         To_The_String  :  in out  STRING_TYPE

      summary:

         The procedure consists entirely of a call to
         to overloaded COPY procedure.  ASSIGN exists
         solely for the use of the MAP component.
```

Figure 5.2. Summary of String Component Modifications

list and a string is that a string also includes a class of operations devoted to manipulating substrings. This class of operations does not exist in the list component. The implementation of a list component is that of the

singly-linked, unbounded, unmanaged list as presented by Booch [1987: 79-85].

While the high-level description of the object model refers to sets of attributes, behaviors, and objects, and since there is a standard reusable component SET, the natural tendency was to use the SET component. However, the key problem with using Booch's SET component was that it is (in Booch's idiom) a monolithic structure, that is, "the structure is always treated as a single unit and that individual parts of the structure can not be manipulated" [Booch, 1987: 38]. Thus, while Booch provides a capability to **REMOVE** an item from a set, and the capability to test whether a particular item **IS_A_MEMBER** of a given set, he provides no capability to take an item from its set, manipulate it, and then return it. Since I clearly wanted to manipulate individual parts of these structures, I required a polylithic structure. I chose not to violate Booch's abstraction, and so examined the provided polylithic structures for an adequate substitute. Of the three polylithic structures provided: lists, trees, and graphs, I selected lists because of their conceptual simplicity, small code bulk, and sufficiency. If, however, I had anticipated large sets, I would have used trees.

The single problem with the list component was that it lacked a deletion or **REMOVE_ITEM** constructor. Booch's argument is that this operation can be built up out of the primitive constructs he provides. However, the use of some of these constructs, notably the **SWAP_TAIL** constructor, is so counterintuitive that it is conceptually simpler to appeal to the underlying structure of the list without layering the fundamental constructs. Figure 5.3 presents a summary of the list component modifications.

```
List_Single_Unbounded_Unmanaged

ADD:

   procedure  Remove_Item

      parameters:

         The_Item       :  in      Item
         From_The_List  :  in out  LIST_TYPE

      summary:

         If  The_Item  is the first item in From_The_
         List, then return the tail of From_The_List,
         else Remove_Item The_Item From_The_List the
         tail of From_The_List.  CONSTRAINT_ERROR is
         raised if From_The_List is NULL, and means
         that The_Item is not in From_The_List.
```

Figure 5.3. Summary of List Component Modifications

5.1.3. Map

The **MAP** is the most complicated reusable component in the system. According to Booch, a map is "a function on object of one type, called the **domain**, yielding objects of a second type, called the **range**" [Booch, 1987: 213]. The purpose of the map in this thesis is to provide some functional dynamic binding capability to accompany the data dynamic binding capability implicit in the object model. The particular form of the map component used in this research was that of a simple, noncached, sequential, unbounded, unmanaged, noniterator map.

The details of the systemic use of the map component are provided under the description of the Operation_Package (section 5.3.3). The design and implementation use of this reusable component was fraught with difficulty. The main problem with the provided map component was that my domain type was a string type (section 5.1.1), which is a **limited** type. The map component assumes that the domain is not a limited type, that is, it assumes that the assignment operator (:=), and the tests for equality (=) and inequality (/=) are available. Given this disparity

between the assumptions underlying the provided map component and the objective reality of the characteristics of the domain type in the system, substantial modifications to the map component had to be undertaken.

These modifications occurred at three levels. The first level was the modification to the formal generic type parameter DOMAIN_TYPE. Figure 5.4 summarizes the Map component DOMAIN_TYPE modification.

```
+-------------------------------------------------------------+
|                                                             |
|    Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_     |
|        Iterator                                             |
|    MODIFY:                                                  |
|       generic type parameter:  DOMAIN_TYPE                  |
|    change from:  private                                    |
|    change to  :  limited private                            |
|                                                             |
+-------------------------------------------------------------+
```

Figure 5.4. Map Component DOMAIN_TYPE Modification Summary

Following from the change from a private to a limited private DOMAIN_TYPE was the additions of the generic function "=" and procedure ASSIGN. These additions were required to replace the implicit "=" function and ":=" function of the private DOMAIN_TYPE. Figure 5.5 summarizes the Map component additions.

```
+-------------------------------------------------------------+
|    Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_     |
|        Iterator                                             |
|    ADD:                                                     |
|                                                             |
|    1. generic function:     "="                             |
|          parameters:  Left   :  in       DOMAIN_TYPE        |
|                       Right  :  in       DOMAIN_TYPE        |
|                       returns  :  BOOLEAN                   |
|                                                             |
|    2. generic procedure:   Assign                           |
|          parameters:  The_Domain     : in     DOMAIN_TYPE   |
|                       To_The_Domain : in out DOMAIN_TYPE   |
+-------------------------------------------------------------+
```

Figure 5.5. Map Component Additions Summary

The original **Bind** and **Copy** procedures had to be modified to reflect the use of the **Assign** procedure. Additionally, the use of aggregate assignments by Booch in the original had to be curtailed, since their use is illegal with limited private types. Procedure Bind is depicted in Figure 5.6.

```
+-------------------------------------------------------------+
|    Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_     |
|        Iterator                                             |
|                                                             |
|    procedure  Bind                                          |
|                                                             |
|    change from:                                             |
|       The aggregate assignment for In_The_Map(The_Bucket)   |
|       to the new NODE_TYPE.                                 |
|                                                             |
|    change to:                                               |
|       The use of the Assign procedure to initialize         |
|       In_The_Map(The_Bucket)'s new NODE_TYPE's Domain.      |
+-------------------------------------------------------------+
```

Figure 5.6. Map Component Modifications: procedure Bind

Finally, two modifications were required for the procedure Copy. These modifications are the same as were required for the Bind procedure above, and are shown in Figure 5.7.

Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator

procedure Copy

Change 1.

change from:

The aggregate assignment for To_The_Map(Index) to the new NODE_TYPE.

change to:

The use of the Assign procedure to initialize To_The_Map(Index) new NODE_TYPE's Domain, and include item-by-item initializations of the other NODE_TYPE fields.

Change 2.

change from:

The aggregate assignment for To_Index.Next to the new NODE_TYPE.

change to:

The use of the Assign procedure to initialize To_Index.Next's new NODE_TYPE's Domain, and include item-by-item initializations of the other NODE_TYPE fields.

Figure 5.7. Map Component Modifications: procedure Copy

5.2 Object Model

As described in Chapter 4, the object model itself is comprised of a Utilities package, an Attribute package, a Behavior package, and an Object package, and this shall be the order of presentation for the object model components. An alternative structure would have the Attribute, Behavior, and Utilities packages as sub-packages within the Object package, however, it was in the interest of this thesis that the structure be kept as open and flexible as possible.

While conceptual simplicity might dictate that the object model packages export only their respective types, procedural exports were also allowed. These procedural exports deal with the storage, retrieval, and display of the objects and their constituent parts. These procedures should actually be CAIS-oriented, allowing transportability of the system across different architectures.

5.2.1 Utilities

The Utilities package exports a miscellanea of types and procedures which are mainly concerned with the manipulation, storage, retrieval, and display of variable

length character strings. Thus a more appropriate name for this package might have been String_Utilities. Even so, it is the intent of this chapter to document the current state of the system, not necessarily its likely improvements. Such information is contained in Chapter 7. The Utilities package is summarized in Figure 5.8.

```
+-------------------------------------------------------------+
|   package:  Utilities                                       |
|                                                             |
|   external references:                                      |
|      Text_IO                                                |
|      String_Sequential_Unbounded_Unmanaged_Noniterator      |
|                                                             |
|   exports:                                                  |
|                                                             |
|      instantiations:                                        |
|                                                             |
|         Natural_IO                                          |
|         Character_String                                    |
|                                                             |
|      variables, types, and constants:                       |
|                                                             |
|         DEBUG  -  a BOOLEAN constant which acts as a        |
|                   switch enabling periodic dumps during     |
|                   program execution.                        |
|                                                             |
|      procedures and functions:                              |
|                                                             |
|         Hash              -  function                      |
|         Store_String      -  procedure                     |
|         Scan_Past_Next    -  procedure                     |
|         Display_String    -  procedure                     |
|         Retrieve_String   -  function                      |
|                                                             |
|      exceptions:                                            |
|                                                             |
|         Attempted_to_Hash_a_NULL_String                     |
+-------------------------------------------------------------+
```

Figure 5.8. Utilities Package Summary

5.2.2 Attributes

Since an attribute can be a component of an object, a behavior, or even another attribute, the attribute package is used by every other higher level package in the system. The direct affectation of an object's attributes by objects' behaviors is a fundamental concept in this system and leads to an open structure for the attributes. The attribute package summary is depicted in Figure 5.9.

```
+-------------------------------------------------------------+
|   package:  Attribute_Package                               |
|                                                             |
|   external references:                                      |
|      Text_IO                Utilities                       |
|      List_Single_Unbounded_Unmanaged                        |
|      String_Sequential_Unbounded_Unmanaged_Noniterator      |
|                                                             |
|   exports:                                                  |
|      instantiations: Attribute_List                         |
|      variables, types, and constants:                       |
|         ATTRIBUTE          -  ATTRIBUTE_RECORD pointer      |
|         ATTRIBUTE_RECORD   -  record which consists of      |
|                     Identifier          (string)            |
|                     Parent              (ATTRIBUTE)         |
|                     Value               (string)            |
|                     List_of_Attributes (Attribute_List)     |
|      procedures and functions:                              |
|         Find_Attribute           - function                 |
|         Store_Attribute_List     - procedure                |
|         Store_Attribute          - procedure                |
|         Display_Attribute_List   - procedure                |
|         Display_Attribute        - procedure                |
|         Retrieve_Attribute_List  - function                 |
|         Retrieve_Attribute       - function                 |
+-------------------------------------------------------------+
```

Figure 5.9. Attribute Package Summary

5.2.3 Behaviors

Since a behavior can be a component of an object, or another behavior, the behavior package is used by other high-level packages in the system. The direct affectation of an object's attributes by objects' behaviors is a fundamental concept in this system and leads to an flat structure for the behaviors. The behavior package summary is shown in Figure 5.10.

```
+-------------------------------------------------------------+
|   package:  Behavior_Package                                |
|                                                             |
|   external references:                                      |
|      Text_IO                Utilities                       |
|      List_Single_Unbounded_Unmanaged                        |
|      String_Sequential_Unbounded_Unmanaged_Noniterator      |
|      Attribute_Package                                      |
|                                                             |
|   exports:                                                  |
|      instantiations:  Behavior_List                         |
|      variables, types, and constants:                       |
|         BEHAVIOR          -  BEHAVIOR_RECORD  pointer       |
|         BEHAVIOR_RECORD   -  record which consists of       |
|                     Identifier         (string)             |
|                     Parent             (BEHAVIOR)           |
|                     List_of_Attributes (Attribute_List)     |
|                     List_of_Behaviors  (Behavior_List)      |
|      procedures and functions:                              |
|         Find_Behavior          - function                   |
|         Store_Behavior_List    - procedure                  |
|         Store_Behavior         - procedure                  |
|         Display_Behavior_List  - procedure                  |
|         Display_Behavior       - procedure                  |
|         Retrieve_Behavior_List - function                   |
|         Retrieve_Behavior      - function                   |
+-------------------------------------------------------------+
```

Figure 5.10. Behavior Package Summary

5.2.4 Objects

Since an object can be a component of another object, the object package is used by the other high-level packages in the system. The object herein presented is a complete object and accurately reflects the concepts discussed in Chapter 3. The object package is summarized in Figure 5.11.

```
+-------------------------------------------------------------+
|   package:  Object_Package                                  |
|                                                             |
|   external references:                                      |
|      Text_IO                Utilities                       |
|      List_Single_Unbounded_Unmanaged                        |
|      String_Sequential_Unbounded_Unmanaged_Noniterator      |
|      Attribute_Package                                      |
|      Behavior_Package                                       |
|                                                             |
|   exports:                                                  |
|      instantiations:  Object_List                           |
|      variables, types, and constants:                       |
|         OBJECT            -  OBJECT_RECORD    pointer       |
|         OBJECT_RECORD     -  record which consists of       |
|                     Identifier          (string)            |
|                     Parent              (OBJECT)            |
|                     List_of_Attributes (Attribute_List)     |
|                     List_of_Behaviors  (Behavior_List)      |
|                     List_of_Objects    (Object_List)        |
|      procedures and functions:                              |
|         Find_Object           - function                    |
|         Store_Object_List     - procedure                   |
|         Store_Object          - procedure                   |
|         Display_Object_List   - procedure                   |
|         Display_Object        - procedure                   |
|         Retrieve_Object_List - function                     |
|         Retrieve_Object       - function                    |
+-------------------------------------------------------------+
```

Figure 5.11. Object Package Summary

5.3 HOOKE

As described in Chapter 4, the Hierarchical Object-Oriented Kernel Environment (HOOKE) provides the user an opportunity to manipulate objects directly. Its purpose is to facilitate experimentation with the object model directly in an interactive, albeit crude, environment.

The HOOKE is composed of a VT_100 package, the Hooke package, an Operations package, and the MAIN routine. This is the order in which they are described.

5.3.1 Hooke_Package

The Hooke package, summarized in Figure 5.12, provides most of the basic object manipulation capabilities. Even though the enumerated type HOOKE_Operations includes the items execute and utilize, the acompanying procedures are not implemented in the HOOKE_Package, but are included instead in the Operations_Package. The rationale for this decision is discussed in section 5.3.3.

```
+-------------------------------------------------------------+
|  package:  HOOKE_Package                                    |
|                                                             |
|  external references:                                       |
|     Text_IO                    VT_100                       |
|     Utilities                  Attribute_Package            |
|     Behavior_Package           Object_Package               |
|     Map_Simple_Noncached_Sequential_Unbounded_              |
|        Unmanaged_Iterator                                   |
|                                                             |
|  exports:                                                   |
|                                                             |
|     instantiations:  HOOKE_IO                               |
|                                                             |
|     variables, types, and constants:                        |
|        HOOKE_Operations  -  enumerated type                 |
|           nop,help,create,fetch,modify,destroy,             |
|           examine,utilize,store,execute,quit                |
|                                                             |
|     procedures and functions:                               |
|        Menu                  - procedure                    |
|        Help                  - procedure                    |
|        Create                - procedure                    |
|        Fetch                 - procedure                    |
|        Modify                - procedure                    |
|        Destroy               - procedure                    |
|        Examine               - procedure                    |
|        Store                 - procedure                    |
+-------------------------------------------------------------+
```

Figure 5.12. HOOKE Package Summary

5.3.2. Operation_Package

The operation package, summarized in Figure 5.13, is conceptually the most complex in the system. Its main use is to provide limited dynamic binding for operations. The fundamental operations for a given system must be implemented, but those operations can then be given aliases and/or combined in other various ways to produce a more

complicated operation (built up of sub-operations) by using the HOOKE.

Two HOOKE operations are included in the Operations Package. They are the procedures Execute and Utilize. Execute is included in the Operation Package because it is used to activate the underlying procedures as specified by the user, and thus must have access both to the OPERATION_TYPE and OPERATION_TASK_TYPE. Similarly, since Utilize changes the context in which operations take place it is implemented here also.

While it seems a violation of good software engineering practices to remove the HOOKE *operations* Execute and Utilize from the HOOKE_Package, it was done with good reason. Since the Operation_Package uses the HOOKE_Package, the HOOKE_Package could not use the Operation_Package, because such cyclic context calls are not allowed.

Alternatively, the HOOKE_Package and Operations_Package could have been combined -- producing one large package. However, the Operations_Package is, and should be, separate from the HOOKE operations since the user-implemented operations (behavior primitives) also must reside in the

Operations_Package. Thus the Operations_Package also serves to help isolate the HOOKE from user modifications.

```
package:  Operation_Package

external references:
     Text_IO                        VT_100
     Utilities                      HOOKE_Package
     Attribute_Package              Behavior Package
     Object_Package
     Map_Simple_Noncached_Sequential_
         Unbounded_Unmanaged_Iterator

  exports:
     instantiations:                Operation_Map
     variables, types, and constants:

         The_Operation_Map      -   Operation_Map.MAP_TYPE

         OPERATION_TYPE         -   enumerated type

         OPERATION_TASK_TYPE    -   task type
              The_Operation :  in       OPERATION_TYPE
              The_Object    :  in out   OBJECT

         OPERATION_ACCESS_TYPE  -  access type

     procedures and functions:

        Initialize_Operation_Map  -  procedure
        Execute                   -  procedure
        Utilize                   -  procedure

        Additionally, one procedure should be provided
        for each item in the enumerated type: OPERATION_
        TYPE.  Each procedure provided for this purpose
        must include the parameter:

           The_Object :  in out  OBJECT

     exceptions:    none
```

Figure 5.13 Operations Package Summary

5.3.3 MAIN

The parameterless procedure MAIN is not surprisingly the main routine. MAIN differs from Utilize of Operations_Package only in the fact that it must first create a (vacuous) object to utilize. MAIN is summarized in Figure 5.14.

```
+-------------------------------------------------------------+
|   procedure:  MAIN                                          |
|                                                             |
|   external references:                                      |
|                                                             |
|      Text_IO                                                |
|      Utilities                                              |
|      Attribute_Package                                      |
|      Behavior_Package                                       |
|      Object_Package                                         |
|      HOOKE_Package                                          |
|      Operation_Package                                      |
|                                                             |
|   exports:                                                  |
|                                                             |
|      instantiations:  none                                  |
|                                                             |
|      variables, types, and constants:                       |
|                                                             |
|         The_Current_Context   :  Object_Package.OBJECT      |
|                                                             |
|         The_User_Selection    :  HOOKE_Package.            |
|                                  HOOKE_Operations           |
|                                                             |
|      procedures and functions:  none                        |
|                                                             |
|      exceptions:                none                        |
|                                                             |
+-------------------------------------------------------------+
```

Figure 5.14 Procedure Main Summary

6. CASE STUDY

As proven in section 3.4, the defined object model can simulate a Turing machine. Since the object model software system was designed to be a faithful representation of the Chapter 3 concepts, an appropriate proof of concept for the object model which underlies the HOOKE is the simulation of a Turing machine. The proof of concept for the HOOKE, then would be to implement the Turing machine simulation in an interactive fashion using the HOOKE. Since the HOOKE is not quite complete, only the object structure was actually built interactively.

The basic Turing machine with its infinite tape and finite, but arbitrarily large set of states, cannot actually be implemented on a real machine with finite memory. However, this is a hardware limitation, not a limitation of the model, per se. Given an infinite-memory computer with an Ada compiler, a faithful representation of the defined object model can simulate any Turing machine. This is the significance of the proof of section 3.4.

This case study is not claimed to be exhaustive. Indeed, due to the simplicity of the Turing machine, the finer points of what we called object structured design in Chapter 3 could not be demonstrated, since in this case, the

objects of discernment, objects of representation, and objects of execution are all the same. And, finally, in terms of the object classification taxonomy, the only type exercised by this case study is the "closed" object since the move behavior is a behavior of a Turing machine taken against itself.

6.1 Design

There are two key elements to the Turing machine simulation. First, the Turing machine object itself is required. This is the part which can be built interactively on the HOOKE, but which suffers from hardware limitations. The second element, the MOVE procedure, is an example of what has been described as a primitive operation and must be built offline from the HOOKE. Once built, though, the MOVE procedure becomes a part of the HOOKE through its OPERATION_TYPE element. Behaviors can then be bound to this operation through the Operation_Map.

6.1.1. The Basic Turing Machine Object

The proof of section 3.4 provides most of the information needed for the Turing machine development under the HOOKE. The most critical point is that since the HOOKE provides not only the capability to statically represent

information, or knowledge, but also the capability to process the data and its structure, the object structure, must be well defined for the primitive operation(s) to be effective. Initially, while the system is being structured, the behavior MOVE could be bound to the primitive operation nop as the equivalent of "stubbing out" the move behavior.

6.1.1.1 Define the Object

The following is taken from the proof of section 3.4.

A Turing machine object consists of:
- A tape head object which consists of:
  - a location attribute and
  - a current input attribute,
- an input tape object which consists of:
  - a set of tape cell objects. Each tape cell object consists of:
    - a location attribute and
    - a tape symbol attribute

E a special tape cell object called the leftmost cell, the value of whose location attribute is 0.

Each tape cell object may hold exactly one of a finite set of allowable tape symbols (gamma) in its tape symbol attribute.

Initially, the n leftmost tape cell objects, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input symbols (sigma).

The remaining infinity of tape cell objects each hold the blank (B), which is a special tape symbol that is not an input symbol.

a finite control object which consists of:
a current state attribute whose value is one of the finite set of states (Q).

a move behavior which consists of
depending on the tape head object's current input attribute and the finite control object's current state attribute:
changes the finite control object's current state attribute to another state,
changes the tape object's tape cell object whose location attribute is the same as the tape head object's location attribute, and
changes the tape head object's location attribute by +/- 1.

The move behavior defined above is an articulation of the next move function (delta) which is a mapping from Q x gamma to Q x gamma x { L , R } where delta may be undefined for some arguments (the tape head object's current input attribute and the finite control object's current state attribute).

6.1.1.2 Explicitly Annotate the Attribute, Behavior, and Object Existential Information

From the above text, we further reformat and add an explicit reference to each defined object model subcomponent (attribute, behavior, or object) if that subcomponent fails to exist. In the following, the original (section 3.4/section 6.1.1.1) text is boldface.

A Turing machine object consists of

- no attributes
- a move behavior
- a tape head object which consists of

-- a location attribute and
a current input attribute),
-- no behaviors
-- no objects

an input tape object which consists of

-- no attributes
-- no behaviors
-- a set of tape cell objects, each of which consist of

--- a location attribute and
a tape symbol attribute
--- no behaviors
--- no objects

E a special tape cell object called the leftmost cell, the value of whose location attribute is 0.

Each tape cell object may hold exactly one of a finite set of allowable tape symbols (gamma) in its tape symbol attribute.

Initially, the n leftmost tape cell objects, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input symbols (sigma).

The remaining infinity of tape cell objects each hold the blank (B), which is a special tape symbol that is not an input symbol.

a finite control object which consists of

-- a current state attribute whose value is one of the finite set of states (Q)
-- no behaviors
-- no objects

6.1.1.3 Compress and Format the Object

Finally, we take out all of the non-essential words

leaving only the object, attribute, and behavior names,

which we enclose in backslashes which we denote here as slashes (/). Where we specified no attributes, behaviors, or objects above we substitute "{}".

```
[ /Turing machine/
  {}
  { [ /move/ {} {} ] }
  { [ /tape head/
      {
        [ /location/       // {} ]
        [ /current input/ // {} ]
      }
     {}
     {}
   ]
  [ /input tape/
     {}
     {}
     {
       [  /tape cell i/
          {
            [ /location/     // {} ]
            [ /tape symbol/ // {} ]
          }
          {}
          {}
        ]
     }
  ]
```

E a special tape cell object called the leftmost cell, the value of whose location attribute is 0.

Each tape cell object may hold exactly one of a finite set of allowable tape symbols (gamma) in its tape symbol attribute.

Initially, the n leftmost tape cell objects, for some finite $n \geq 0$, hold the input, which is a string of symbols chosen from a subset of the tape symbols called the input symbols (sigma).

The remaining infinity of tape cell objects each hold the blank (B), which is a special tape symbol that is not an input symbol.

```
        [ /finite control/
          {
            [ /current state/ /qi/ {} ]
          }
          {}
          {}
        ]
      }
    ]
```

It is important to note that with the above minor structural modifications to the proof, the above format is directly readable by the HOOKE (fetch operation). The embedded text concerning certain tape cell information will be ignored as comments.

6.1.2 The Delta Function

The delta function is defined as a "mapping from Q x gamma to Q x gamma x { L, R } (delta may, however, be undefined for some arguments)" [Hopcroft and Ullman, 1979: 148]. Within the HOOKE we can represent the delta function as an object:

```
[ /delta/ - object identifier

  {}      - set of attributes (empty)
  {}      - set of behaviors  (empty)
  {       - set of objects
```

```
[ /q0/  - a given state from Q (defines a state
          transition table row)
  {}    - set of attributes (empty)
  {}    - set of behaviors  (empty)
  {     - set of objects

    [  /0/  -  a given input symbol from gamma (defines
               a state transition table column)
       {    -  set of attributes

         [ /next state/   /qn/ {} ] - a state from Q
         [ /output symbol/ /1/ {} ] - a symbol from
                                      gamma
         [ /direction/     /L/ {} ] - tape head
                                      direction
       }
      {}   - set of behaviors (empty)
      {}   - set of objects   (empty)
    ]
    .
    .  - one object per state transition table entry
    .
  }
 ]
 .
 .  - one object per state transition table row
 .
}
]
```

Although the above object is a state transition table stored in row-major format, it would have been just as easy to have stored it in column-major. The essential point is that the MOVE must be built in accordance with the delta representation.

6.1.3 The Input Tape

Finally, although the input tape is a simple linear structure composed of cells, the structure under the object model is more complicated than a simple array. This is due to the general nature of the object model wherein no ordering information is implicit structurally. Therefore, explicit ordering information must be carried with each cell as an attribute.

```
[ /input tape/ - object identifier

  {}      - set of attributes (empty)
  {}      - set of behaviors  (empty)
  {       - set of objects

    [ /0/ - leftmost tape cell
      {
        [ /tape symbol/ /1/ {} ]
      }
      {}    - set of behaviors  (empty)
      {}    - set of objects    (empty)
    ]

    [ /1/
      {
        [ /tape symbol/ /B/ {} ]
      }
      {}    - set of behaviors  (empty)
      {}    - set of objects    (empty)
    ]

      .
      .  - one tape cell object for each tape cell
      .
    }
]
```

6.1.4 The MOVE Behavior

The final major element is the MOVE primitive operation. Structurally, it will be represented as a single parameter (in out Object) procedure. The primitive operations all operate directly on objects and their underlying components. To do that, though, the object structure must be known to the procedure.

The MOVE procedure code must fit into the following template, where it is assumed that The_Object being sent is the Turing machine, itself:

```
procedure
   procedure_name
   (
      The_Object : in out Object_Package.OBJECT
   )
   is
       begin
          { user-written statements }
       end  procedure_name;
```

The MOVE pseudocode follows:

```
The_Current_State          := The_Object.
                              Finite_Control.
                              The_Current_State
```

```
The_Current_Location      := The_Object.
                             Tape_Head.
                             Location

The_Current_Input_Symbol := The_Object.
                             Input_Tape(The_Current_Location).
                             Tape_Symbol

Transition_Cell           := The_Object.
                             Delta.
                             The_Current_State.
                             The_Current_Input_Symbol

if    The_Object_List_Is_Null  --  delta(q,g) maps to nil
then
      print (reject) message and halt
else
      The_Next_State    := Transition_Cell.The_Next_State
      The_Output_Symbol := Transition_Cell.The_Output_Symbol
      The_Direction     := Transition_Cell.The_Direction

      The_Current_State := The_Next_State

      The_Object.
      Finite_Control.
      The_Current_State := The_Current_State

      The_Object.
      Input_Tape(The_Current_Location).
      Tape_Symbol       := The_Output_Symbol

      if    The_Direction = L
      then

         if    The_Current_Location = 0
         then
            print (reject) message and halt  -- tried to
                                             -- scan left
                                             -- off tape
         else
            The_Current_Location := The_Current_Location -1
         end if;

      else  --  The_Direction = R

         The_Current_Location := The_Current_Location + 1

      end if;
```

```
        The_Object.Tape_Head.Location := The_Current_Location

        An_Object := The_Object.
                     Input_Tape(The_Current_Location)

        if    The_Object_List_Is_Null -- that is, scanning the
                                         leftmost tape cell of
                                         the infinity of blank
                                         cells on the right
                                         portion of the input
                                         tape
        then  --  build a blank tape cell

              The_New_Tape_Cell  :=  new  OBJECT_RECORD;

              The_New_Tape_Cell.
              Identification     := The_Current_Location

              Clear The_New_Tape_Cell.List_of_Attributes

              The_New_Tape_Symbol := new Attribute

              The_New_Tape_Symbol := B -- the blank

              Add The_New_Tape_Symbol to
                  The_New_Tape_Cell.List_of_Attributes

              Clear The_New_Tape_Cell.List_of_Behaviors

              Clear The_New_Tape_Cell.List_of_Objects

              Add The_New_Tape_Cell to
                  The_Object.Input_Tape.List_of_Objects
        end if;

        The_Object.
        Tape_Head.
        The_Current_Input := The_Object.
                             Input_Tape(The_Current_Location).
                             Tape_Symbol

        if    The_Current_State in The_Object.Q.Final
        then
              print (accept) message and halt
        end if;

end if;
```

6.2. Execution

Several different delta functions were generated against which several different input tapes were applied. In all cases the results were as expected, that is, in those cases where the Turing machine is predicted to accept, it does so; and in those cases where the Turing machine is predicted to fail to accept, it does so i. the correct fashion, e.g. by attempting an undefined transition. The actual test cases and their results are summarized in Appendix B.

The execution of the Turing machine simulation, although predictably slow, is correct. The simulation responds just as a real Turing machine would given the same delta function and input tape. The examples used were admittedly simple. There were three reasons for this. First, the examples were intended to provide a proof of concept, not an exhaustive test suite. Second, examples were chosen whose results could be relatively easily hand-checked. And finally, since the structure of the object model and the Turing machine (including the delta function and the input tape) are very regular, testing requirements are reduced.

6.3 Conclusion

Since we have already proved that the object model can simulate a Turing machine, and therefore represent any computable function, our case study results have to do with this particular implementation. We conclude that the object model implementation is correct, and that the HOOKE functions well enough to warrant further development. The worst aspect of the system is the clumsy method required to establish a new user-defined primitive operation. However, since the system is written entirely in Ada, which lacks the dynamic binding capabilities of languages like Smalltalk, that could not be helped.

7. Conclusions and Recommendations

This chapter first summarizes how the research conducted and software developed met the goals established in the problem statement. It next presents conclusions drawn from the work, provides specific recommendations for the enhancement of the environment and the object model as implemented, and outlines future work suggested by these results. Finally, it explores a few of the implications of the key results.

7.1 Summary

The primary goal was to provide a definition of an object model and consider its theoretical foundations. We took the approach of defining an object in its abstract sense before attempting implementation, instead of defining it by its implementation. Once we defined the model, we proved it can simulate a Turing machine.

The next most important goal for this research was to actually implement the object model in Ada, in order to empirically investigate its properties. While the model implemented was a faithful representation of the definition

advanced and proven in Chapter 3, it is not claimed to be the best possible representation of the object model.

The third subgoal was to implement a prototype environment that would allow a user to directly and interactively manipulate the object model. The environment does, in fact, provide some direct, interactive access to the object model as was shown by building the object structure of a Turing machine simulation interactively. Thus all of the stated research goals were met, although it would have been more gratifying to have completed the environment -- allowing interactive access to the object model's behaviors and attributes.

7.2 Conclusions

The most significant conclusion that can be drawn from this work is that the defined object model is a theoretically sound basis for representing programs. That is, the model is complete in the sense that whatever we can program, we can represent under the object model. The significance of this result is that the object model then provides a consistent mode of representation across all programming problems. In general, then, since the object

model provides a base abstraction for the object-oriented paradigm, and since the object model is equivalent to a Turing machine, then the object-oriented paradigm is capable of representing any program.

The answer to the question of whether Ada is or is not an object-oriented programming language is yes, to a degree. While this question was not addressed specifically in this research, the general thrust of the question is the whole point of the research. The point is that the whole question of what constitutes an object-oriented programming language begs the question of whether the object-oriented paradigm is viable. This we have already answered in the affirmative. Since all programming languages have object-oriented features to some extent, we can say that Ada is more object-oriented than Assembler or Fortran IV, but less object-oriented then CLU or Smalltalk.

Unexpectedly, an additional conclusion can be derived from the difficulties experienced using the reusable components which were detailed in Chapter 5. Currently, there are at least two problems with reusable software components. First, the component author's abstraction must be thoroughly understood and therefore completely

articulated before the component can be effectively used. And, second, the number of different components necessary in Ada do not reflect solely logical, structural, and operational classifications, but must also account for artifacts of the language, for example using a limited type as the domain for a map component.

7.3 Recommendations

Work remains to be done on the object model implementation. First, the object model implementation should be converted into an abstract data type. To accomplish this, the Attribute_Package and Behavior_Package should be embedded in the Object_Package, and the String package should be instantiated directly into the Object_Package as should the Operation_Map, which would have to be provided a specific Operation_Type enumerated type by the client of the Object_Package.

The Hierarchical Object-Oriented Kernel Environment (HOOKE) also needs work. Particularly, the Modify, Execute, and Help procedures need to be completed. Also, based on

the changes recommended for the object implementation above, the environment must be adapted to the new object implementation.

7.4 Future Work

On a theoretical level, empirical investigations of the object model should continue. In particular, different implementations of the behavior/primitive operation mapping should be studied, as should efficiency issues. Additionally, the meta-behaviors of burst (i.e. execute all behaviors in parallel), sequence, iteration, and selection should be added to the execution of an object's set of behaviors. The prototype environment provides a rudimentary capability to directly experiment with the capabilities of the defined object model. For the environment, a graph-like structure for relating the objects should be investigated instead of the current virtual "n-way tree" structure.

From the system development viewpoint, the general question of the viability of the object model as a knowledge representation paradigm was not investigated in sufficient detail to allow any conclusions to be drawn. Since the naturalness of decomposition under the object-oriented

paradigm is considered to be one of the most useful features of object-oriented programming languages, we conjecture that the object model as defined appears to be capable of representing the real world. The full breadth of interactive object construction was not explored. This includes the exploration of concepts like interactive system decomposition. However, since the capability to directly and interactively manipulate an object and its subobjects, create new objects, examine, save, restore, and destroy objects, all proved useful, the broader concept of also interactively manipulating the attribute and behavior components of an object is worthy of further investigation.

On a practical level, the object model should be coupled with a relatively simple Eliza-like interface to produce an object-oriented problem definition and requirements analysis program. For example, when a user states a need for something, the system checks its current context for an object with that identifier. If the object already exists, the system offers it to the user. The user can accept it, reject it, or ask to modify it. If the object does not exist in the current context, the system should check other contexts. If the object is found, a temporary clone is made and offered to the user who can then accept

it, reject it, or modify it. If no object can be found, then the system asks for a description, parses the description and searches for the components of the description in much the same fashion as outlined. The output from this interview-type human-machine interaction would then produce an object-oriented concept map from which a hierarchy of requirements are derivable.

The object model should also be investigated as a unifying concept for the model base management system component of decision support systems. This is a particularly rich research area which includes not only the integration of different types of models, but also the construction of new models like object-oriented simulations.

Finally, a nomenclature for reusable software components should be defined. Such a nomenclature would probably not be able to be both short (usable) and descriptive, but a short usable nomenclature could be developed keyed to a descriptive notation which would be consistent across components so that their properties could be compared. Such a descriptive notation, component nomenclature, and classification scheme should be developed.

7.5 Implications

The theoretical foundation of the object-oriented paradigm is securely anchored in computability theory. A design process based on this paradigm has the advantage of provable completeness; thus, a consistent design process can be developed that will be effective across all computer science application areas. While the specific object model proposed in this thesis can only prove its worth over time, the model provides a basic abstraction upon which to build an object-oriented design process.

A. Pseudocode

This appendix contains the pseudocode for the modules described in the detailed design presented in Chapter 5. The modules are arranged in the same order they are referenced in Chapter 5. The reusable components are not included here since they are adequately described in Chapter 5 and their source, e.g. Booch [1987].

A.1 Object Model

The following pseudocode describes the Utilities package, Attribute package, Behavior package, and Object package.

A.1.1 Utilities

The Utilities package is mainly concerned with the manipulation, storage, retrieval, and display of variable length character strings.

A.1.1.1 Hash

Given a variable length character string, Hash returns a positive integer less than the system's Integer'last.

```
IF      THE INPUT STRING IS NULL
THEN    RAISE ATTEMPTED TO HASH A NULL IDENTIFIER
ELSE
        USE A QUADRATIC FOLDING HASH ALGORITHM :

           SET THE CURRENT VALUE TO ZERO

           FOR EACH CHARACTER IN THE INPUT STRING

              TAKE THE INTEGER POSITION OF THE CHARACTER
              IN THE CHARACTER SET - THIS IS A UNIQUE
              STRING OF BITS

              SHIFT THIS STRING OF BITS TO THE RIGHT SO
              THAT FOUR CHARACTERS TAKE UP ONE 32-BIT
              INTEGER NUMBER

              IF    ADDING THAT 32-BIT INTEGER TO THE
                    CURRENT VALUE, WILL CAUSE THE CURRENT
                    VALUE TO EXCEED INTEGER'LAST

              THEN

                    SUBTRACT INTEGER'LAST FROM THE
                    CURRENT VALUE AND THEN ADD THE 32-BIT
                    INTEGER.

              ELSE

                    ADD THE 32-BIT INTEGER TO THE CURRENT
                    VALUE.

          WHEN THE SUPPLY OF CHARACTERS IS EXHAUSTED,
          RETURN THE CURRENT_VALUE
```

A.1.1.2 Store-String

Given a variable length character string and a file, Store_String writes the string into the file.

```
OUTPUT THE CHARACTER STRING DELIMITER ('\')
OUTPUT THE CHARACTER STRING
OUTPUT THE CHARACTER STRING DELIMITER ('\')
```

A.1.1.3 Scan_Past_Next

Given a delimiter of type character and an input file, Scan_Past_Next will read the file, a character-at-a-time until it has read a character matching the delimiter or the end of file.

```
LOOP

   GET A CHARACTER
   IF    THE CHARACTER MATCHES THE DELIMITER
   THEN  EXIT

END LOOP
```

A.1.1.4 Display_String

Given an input string, Display_String Text_IO.PUTs that string to the "file" STANDARD_OUTPUT.

```
OUTPUT THE CHARACTER STRING DELIMITER ('\')
OUTPUT A SPACE (' ') CHARACTER
OUTPUT THE CHARACTER STRING
OUTPUT A SPACE (' ') CHARACTER
OUTPUT THE CHARACTER STRING DELIMITER ('\')
```

A.1.1.5 Retrieve_String

Given an input file, Retrieve_String returns the next string as stored by Store_String (5.2.1.2) from that file.

```
ESTABLISH A NULL TEMPORARY STRING
Scan_Past_Next '\' IN THE FILE

LOOP

   Text_IO.GET A CHARACTER FROM THE FILE

   IF    THE CHARACTER = '\'

   THEN  EXIT

   ELSE
         APPEND THE CHARACTER TO THE INPUT
         STRING

END LOOP

IF   THE INPUT STRING IS NOT NULL

THEN

     COPY THE INPUT STRING TO THE TEMPORARY
     STRING

RETURN THE TEMPORARY STRING
```

A.1.2 Attributes

The attribute package underlies both of the other object model components.

A.1.2.1 Find_Attribute

Given an attribute identifier (character string) and an attribute list, search the list for the attribute. If found return a pointer to the attribute, otherwise return a null pointer.

```
SET AN INDEX TO THE HEAD OF THE ATTRIBUTE LIST

WHILE  (1) THE INDEX IS NOT NULL  AND THEN
       (2) THE ITEM INDICATED BY INDEX IS NOT
           THE DESIRED ATTRIBUTE
LOOP
       SET INDEX TO THE TAIL OF THE LIST
           INDICATED BY INDEX
END LOOP

RETURN INDEX
```

A.1.2.2 Store_Attribute_List

Given a file and an attribute list, Store_Attribute_List will store the attribute list in the file in a format which Retrieve_Attribute_List can retrieve.

```
SET THE INDEX TO HEAD OF THE ATTRIBUTE LIST

Text_IO.PUT '{' INTO THE FILE
             -- START ATTRIBUTE LIST DELIMITER

WHILE THE INDEX IS NOT NULL
LOOP

      Store_Attribute THE HEAD OF THE LIST
         THE INDEX

      SET INDEX TO THE TAIL OF THE LIST
         THE INDEX

END LOOP

Text_IO.PUT '}' INTO THE FILE
             -- END ATTRIBUTE LIST DELIMITER
```

A.1.2.3 Store_Attribute

Given a file and an ATTRIBUTE (a pointer to an ATTRIBUTE_ RECORD), Store_Attribute will store the attribute in the file in a format which Retrieve_Attribute can retrieve.

```
Text_IO.PUT '[' INTO THE FILE
             -- START ATTRIBUTE DELIMITER

Store_String THE ATTRIBUTE IDENTIFIER
Store_String THE ATTRIBUTE VALUE

Store_Attribute_List THE ATTRIBUTE'S LIST OF
   ATTRIBUTES

Text_IO.PUT ']' INTO THE FILE
             -- END ATTRIBUTE DELIMITER
```

A.1.2.4 Display_Attribute_List

Display_Attribute_List differs from Store_Attribute_List only in the file being "written." In the latter case the file is passed to Display_Attribute_List. In the former case, the file is assumed to be STANDARD_OUTPUT, i.e. the display terminal screen. Due to this similarity, the pseudocode is not repeated here.

A.1.2.5 Display_Attribute

The correspondence between Display_Attribute and Store_Attribute is identical to that between Display_Attribute_List and Store_Attribute_List. Thus the pseudocode is not repeated here.

A.1.2.6 Retrieve_Attribute_List

Given a file, Retrieve_Attribute_List will retrieve the next attribute list contained in the file as stored by the Store_Attribute_List procedure.

```
BEGIN WITH AN EMPTY TEMPORARY ATTRIBUTE LIST
```

```
LOOP

   Text_IO.GET INPUT CHARACTER FROM THE FILE

   CASE  THE INPUT CHARACTER  IS

      WHEN  '['     RETRIEVE THE ATTRIBUTE
                    ADD THE ATTRIBUTE TO THE
                        TEMPORARY ATTRIBUTE
                        LIST

      WHEN  '}'     LIST DELIMITER: EXIT

      WHEN  others  SKIP ALL ELSE:  DO NOTHING

   END CASE

END LOOP

RETURN THE TEMPORARY ATTRIBUTE LIST
```

A.1.2.7 Retrieve_Attribute

Given a file, Retrieve_Attribute will retrieve the next attribute contained in the file as stored by the Store_Attribute procedure.

```
BEGIN WITH A NEW TEMPORARY ATTRIBUTE

READ THE ATTRIBUTE IDENTIFIER
READ THE ATTRIBUTE VALUE

Scan_Past_Next '{' -- ATTRIBUTE LIST DELIMITER

Retrieve_Attribute_List INTO THE TEMPORARY
   ATTRIBUTE'S LIST_OF_ATTRIBUTES

Scan_Past_Next ']' -- END ATTRIBUTE DELIMITER

RETURN THE TEMPORARY ATTRIBUTE
```

A.1.3 Behaviors

Behaviors underly objects and depend on attributes. They are not an intermediate form in any case, but a separate structure.

A.1.3.1 Find_Behavior

Given an behavior identifier (character string) and a behavior list, search the list for the behavior. If found, return a pointer to the behavior, otherwise return a null pointer.

```
SET AN INDEX TO THE HEAD OF THE BEHAVIOR LIST

WHILE  (1) THE INDEX IS NOT NULL  AND THEN
       (2) THE ITEM INDICATED BY INDEX IS NOT
           THE DESIRED BEHAVIOR
LOOP
       SET INDEX TO THE TAIL OF THE LIST
           INDICATED BY INDEX
END LOOP

RETURN INDEX
```

A.1.3.2 Store_Behavior_List

Given a file and a behavior list, Store_Behavior_List will store the behavior list in the file in a format which Retrieve_Behavior_List can retrieve.

```
SET THE INDEX TO THE HEAD OF THE BEHAVIOR LIST

Text_IO.PUT '{' INTO THE FILE
             -- START BEHAVIOR LIST DELIMITER

WHILE THE INDEX IS NOT NULL
LOOP

      Store_Behavior THE HEAD OF THE LIST
         THE INDEX

      SET INDEX TO THE TAIL OF THE LIST
         THE INDEX

END LOOP

Text_IO.PUT '}' INTO THE FILE
             -- END BEHAVIOR LIST DELIMITER
```

A.1.3.3 Store_Behavior

Given a file and an BEHAVIOR (a pointer to an BEHAVIOR_RECORD), Store_Behavior will store the behavior in the file in a format which Retrieve_Behavior can retrieve.

```
Text_IO.PUT '[' INTO THE FILE
             -- START BEHAVIOR DELIMITER

Store_String THE BEHAVIOR IDENTIFIER

Store_Attribute_List THE BEHAVIOR'S LIST OF
   ATTRIBUTES

Store_Behavior_List  THE BEHAVIOR'S LIST OF
   BEHAVIORS

Text_IO.PUT ']' INTO THE FILE
             -- END BEHAVIOR DELIMITER
```

A.1.3.4 Display_Behavior_List

Display_Behavior_List differs from Store_Behavior_List only in the file being "written." In the latter case the file is passed to Display_Behavior_List by the calling procedure. In the former case, the file is assumed to be STANDARD_OUTPUT, i.e. the display terminal screen. Due to this similarity, the pseudocode is not repeated here.

A.1.3.5 Display_Behavior

The correspondence between Display_Behavior and Store_ Behavior is identical to that between Display_Behavior_List and Store_Behavior_List. Thus the pseudocode is not repeated here either.

A.1.3.6 Retrieve_Behavior_List

Given a file, Retrieve_Behavior_List will retrieve the next behavior list contained in the file as stored by the Store_ Behavior_List procedure.

```
BEGIN WITH AN EMPTY TEMPORARY BEHAVIOR LIST
```

```
LOOP

   Text_IO.GET INPUT CHARACTER FROM THE FILE

   CASE  THE INPUT CHARACTER  IS

      WHEN  '['
             -- BEHAVIOR DELIMITER
                Retrieve_Behavior THE BEHAVIOR
                ADD THE BEHAVIOR TO THE
                    TEMPORARY BEHAVIOR LIST
      WHEN  '}'
             -- LIST DELIMITER
                EXIT
      WHEN  others
             -- SKIP ALL ELSE (E.G. COMMENTS)
                DO NOTHING
   END CASE

END LOOP

RETURN THE TEMPORARY BEHAVIOR LIST
```

A.1.3.7 Retrieve_Behavior

Given a file, Retrieve_Behavior will retrieve the next behavior contained in the file as stored by the Store_Behavior procedure.

```
BEGIN WITH A NEW TEMPORARY BEHAVIOR

READ THE BEHAVIOR IDENTIFIER

Scan_Past_Next '{'
                -- ATTRIBUTE LIST DELIMITER

Retrieve_Attribute_List
   INTO THE TEMPORARY BEHAVIOR'S
           LIST_OF_ATTRIBUTES
```

```
Scan_Past_Next '{'
                    -- BEHAVIOR LIST DELIMITER

Retrieve_Behavior_List
   INTO THE TEMPORARY BEHAVIOR'S
            LIST_OF_BEHAVIORS

Scan_Past_Next '}'
                    -- END BEHAVIOR DELIMITER

RETURN THE TEMPORARY BEHAVIOR
```

A.1.4 Objects

Finally, an object depends on both the attribute package and the behavior package.

A.1.4.1 Find_Object

Given an object identifier (character string) and an object list, search the list for the object. If found, return a pointer to the object, otherwise return a null pointer.

```
SET AN INDEX TO THE HEAD OF THE OBJECT LIST

WHILE  (1) THE INDEX IS NOT NULL  AND THEN
       (2) THE ITEM INDICATED BY INDEX IS NOT
           THE DESIRED OBJECT
LOOP
       SET INDEX TO THE TAIL OF THE LIST
           INDICATED BY INDEX
END LOOP

RETURN INDEX
```

A.1.4.2 Store_Object_List

Given a file and a object list, Store_Object_List will store the object list in the file in a format which Retrieve_Object_ List can retrieve.

```
SET THE INDEX TO THE HEAD OF THE OBJECT LIST

Text_IO.PUT '{' INTO THE FILE
            -- START OBJECT LIST DELIMITER

WHILE THE INDEX IS NOT NULL
LOOP

      Store_Object THE HEAD OF THE LIST
                   THE INDEX

      SET INDEX TO THE TAIL OF THE LIST
                   THE INDEX

END LOOP

Text_IO.PUT '}' INTO THE FILE
            -- END OBJECT LIST DELIMITER
```

A.1.4.3 Store_Object

Given a file and an OBJECT (a pointer to an OBJECT_RECORD), Store_Object will store the object in the file in a format which Retrieve_Object can retrieve.

```
Text_IO.PUT '[' INTO THE FILE
            -- START OBJECT DELIMITER

Store_String THE OBJECT IDENTIFIER
```

```
Store_Attribute_List THE OBJECT'S
                     LIST OF ATTRIBUTES

Store_Behavior_List  THE OBJECT'S
                     LIST OF BEHAVIORS

Store_Object_List    THE OBJECT'S
                     LIST OF OBJECTS

Text_IO.PUT ']' INTO THE FILE
            -- END OBJECT DELIMITER
```

A.1.4.4 Display_Object_List

Display_Object_List differs from Store_Object_List only in the file being "written." In the latter case the file is passed to Display_Object_List by the calling procedure. In the former case, the file is assumed to be STANDARD_OUTPUT, i.e. the display terminal screen. Due to this similarity, the pseudocode is not repeated here.

A.1.4.5 Display_Object

The correspondence between Display_Object and Store_Object is identical to that between Display_Object_List and Store_Object_List. Thus the pseudocode is not repeated here either.

A.1.4.6 Retrieve_Object_List

Given a file, Retrieve_Object_List will retrieve the next object list contained in the file as stored by the Store_Object_List procedure.

```
BEGIN WITH AN EMPTY TEMPORARY OBJECT LIST
LOOP
   Text_IO.GET INPUT CHARACTER FROM THE FILE
   CASE  THE INPUT CHARACTER  IS
      WHEN  '['
            -- OBJECT DELIMITER
               RETRIEVE THE OBJECT
               ADD THE OBJECT TO THE
                   TEMPORARY OBJECT LIST
      WHEN  '}'
            -- LIST DELIMITER
               EXIT
      WHEN  others
            -- SKIP ALL ELSE
               DO NOTHING
   END CASE
END LOOP
RETURN THE TEMPORARY OBJECT LIST
```

A.1.4.7 Retrieve_Object

Given a file, Retrieve_Object will retrieve the next object contained in the file as stored by the Store_Object procedure.

```
BEGIN WITH A NEW TEMPORARY OBJECT

READ THE OBJECT IDENTIFIER

Scan_Past_Next '{'
                  -- ATTRIBUTE LIST DELIMITER

Retrieve_Attribute_List
   INTO THE TEMPORARY OBJECT'S
            LIST_OF_ATTRIBUTES

Scan_Past_Next '{'
                  -- BEHAVIOR LIST DELIMITER

Retrieve_Behavior_List
   INTO THE TEMPORARY OBJECT'S
            LIST_OF_BEHAVIORS

Scan_Past_Next '{'
                  -- OBJECT LIST DELIMITER

Retrieve_Behavior_List
   INTO THE TEMPORARY OBJECT'S
            LIST_OF_OBJECTS

Scan_Past_Next ']'
                  -- END OBJECT DELIMITER

RETURN THE TEMPORARY OBJECT
```

A.2 HOOKE

The HOOKE is composed of a VT_100 package, the Hooke package, an Operations package, and the MAIN routine. Except for the VT-100 package which is not herein described, this is the order in which they are described.

A.2.2 Hooke_Package

The Hooke package provides most of the basic object manipulation capabilities, excluding execute and utilize.

A.2.2.1 Menu

Menu is the user's main interface to the system. It provides the user with a top-level view of the current context (its set of attributes, behaviors, and objects), and a set of commands with which the user can manipulate the object. Menu includes the procedure Get_Selection in order to localize the interactive I/O.

```
GET SELECTION:

   LOOP

      POSITION THE CURSOR
      PROMPT THE USER
      HOOKE_IO.GET THE_USER_SELECTION

      IF     THE_USER_SELECTION
             IS IN HOOKE_OPERATIONS
      THEN   RETURN  THE_USER-SELECTION
      ELSE

             IF    DATA ERROR
                   (INVALID SELECTION)
             THEN PRINT CONDOLENCES
                   TRY AGAIN

             END IF
```

```
        END IF

    END LOOP

END GET_SELECTION


CLEAR SCREEN
DISPLAY CURRENT CONTEXT
LABEL THE ATTRIBUTE, BEHAVIOR, AND OBJECT
   COLUMNS
DISPLAY THE LIST OF AVAILABLE COMMANDS

DISPLAY COLUMN OF ATTRIBUTES
DISPLAY COLUMN OF BEHAVIORS
DISPLAY COLUMN OF OBJECTS

GET SELECTION
```

A.2.2.2 Help

This procedure provides context sensitive help. An object (the current context) which has a "user help" attribute will have its user help information displayed when help is requested from the menu. This procedure is not yet implemented.

A.2.2.3 Create

As the name implies, and Chapter 4 states, this procedure isused solely to create new (vacuous) objects. It also adds the new object to the current context's list of objects.

```
GET THE NEW OBJECT'S IDENTIFIER

ATTEMPT TO OPEN THE INPUT FILE
   (FILENAME=IDENTIFIER & ".OBJ")

IF    THE IDENTIFIER IS NON-UNIQUE
THEN  APOLOGIZE AND RETURN
ELSE
      GET A NEW OBJECT RECORD
      GIVE IT THE IDENTIFIER
      ENSURE THE ATTRIBUTE LIST IS CLEAR
      ENSURE THE BEHAVIOR  LIST IS CLEAR
      ENSURE THE OBJECT    LIST IS CLEAR
      ADD THE NEW OBJECT TO THE CURRENT
         CONTEXT'S  OBJECT LIST

END IF
```

A.2.2.4 Fetch

Fetch retrieves an already existing object from the native file system and adds it to the current context's list of objects.

```
GET THE OBJECT'S IDENTIFIER

ATTEMPT TO OPEN THE INPUT FILE
   (FILENAME = IDENTIFIER & ".OBJ")

IF    THE FILE DOESN'T EXIST
THEN  APOLOGIZE AND RETURN
ELSE

      GET A NEW OBJECT RECORD

      Retrieve_Object FROM THE FILE INTO THE
         NEW OBJECT RECORD

      NEW OBJECT PARENT IS THE CURRENT CONTEXT
```

ADD THE NEW OBJECT TO THE CURRENT
CONTEXT'S LIST OF OBJECTS

CLOSE THE INPUT FILE

END IF

A.2.2.5 Modify

This provides the interactive object editing capabilities for the system. An object's attributes or behaviors can be modified. This procedure is currently stubbed out.

A.2.2.5.1 Modify_Attribute

Provides the capability to add or delete an attribute, change an attribute's value, or modify the attributes of an attribute.

A.2.2.5.2 Modify_Behavior

Provides the capability to add or delete a behavior, bind or unbind a behavior to or from an operation, or modify the attributes or behaviors of a behavior.

A.2.2.6 Destroy

Destroy removes an object from the current context's list of objects, effectively deleting the object from the system.

```
GET THE OBJECT'S IDENTIFIER

FIND    THE ITEM WITH THAT IDENTIFIER
        IN THE CURRENT CONTEXT'S LIST OF OBJECTS

REMOVE THE ITEM
        FROM THE CURRENT CONTEXT'S LIST OF
             OBJECTS

IF      THE ITEM IS NOT IN THE LIST OF OBJECTS
        OR THE LIST OF OBJECTS IS NULL
THEN
        APOLOGIZE AND RETURN
```

A.2.2.7 Examine

Examine writes a copy of a given object out to the STANDARD_OUTPUT device.

```
GET THE OBJECT'S IDENTIFIER

IF      THE IDENTIFIER IS "self"
THEN
        THE OBJECT IS THE CURRENT CONTEXT
ELSE
        GET THE OBJECT
            FROM THE CURRENT CONTEXT'S OBJECT
               LIST

Display_Object THE OBJECT
```

```
IF      THE ITEM IS NOT IN THE LIST OF OBJECTS
THEN
        APOLOGIZE AND RETURN
```

A.2.2.8 Store

Store writes a copy of a given object out to the native file system. The filename is built from the string ".OBJ" appended to the object name

```
GET THE OBJECT'S IDENTIFIER

FIND THE OBJECT
     IN THE CURRENT CONTEXT'S LIST OF OBJECTS

IF     FOUND
THEN

       OPEN THE FILE:  IDENTIFIER & ".OBJ"
            FOR OUTPUT

       IF    THE FILE DOES NOT EXIST
       THEN
            CREATE THE FILE:  IDENTIFIER &
                                ".OBJ"
            OPEN    THE FILE
                    FOR OUTPUT
       END IF

       Store_Object THE OBJECT
                     INTO THE FILE

       CLOSE THE FILE

ELSE

       APOLOGIZE AND RETURN

END IF
```

A.2.3. Operation_Package

The operation package conceptually the most complex in the system. Note that two HOOKE operations are included in the Operations Package, the procedures Execute and Utilize.

A.2.3.1 OPERATION_TASK_TYPE

The OPERATION_TASK_TYPE is the key element in providing functional dynamic binding. The single accept statement provides a consistent interface across all primitive operations. The task is sent an OPERATION_TYPE item (extracted from the Operation_Map) and an object upon which to perform that operation. The OPERATION_TYPE item designates the correct user-defined procedure through a case statement which tests for at least some of the OPERATION_TYPE items.

```
LOOP
   ACCEPT
      EXECUTE  THE OPERATION
               ON THE OBJECT
        DO
```

```
CASE   THE OPERATION
IS

  WHEN  OPERATION_TYPE_0  =}
     Operation_Package.OPERATION_TYPE_0
             (
                THE OBJECT
             );
             .
             .
             .

   WHEN  OPERATION_TYPE_N  =}
      Operation_Package.OPERATION_TYPE_N
             (
                THE OBJECT
             );
END CASE
```

A.2.3.1 Initialize_Operation_Map

Initialization of the Operation Map is a straightforward procedure, but one which requires an intermediate procedure, Bind_Substring. Since the Operation_Map's Domain type was the Utilities.Character_String.STRING_TYPE, simple user-defined Standard.String objects, e.g. "move head left", could not be directly bound to the Operation_Package.OPERATION_TYPE items. Therefore, Bind_Substring converts the Standard.String user name to Character_String.STRING_TYPE which could then be bound.

```
BIND_SUBSTRING:

   Character_String.Copy   THE SUBSTRING
                           TO THE TEMPORARY
                           STRING

   Operation_Map.Bind      THE TEMPORARY STRING
                           TO THE OPERATION
                           IN THE MAP

END  BIND_SUBSTRING


BIND_SUBSTRING    THE SUBSTRING
                       "{user title}"
                  TO THE OPERATION
                       OPERATION_TYPE_0
                  IN THE MAP
                       OPERATION MAP

                  .
                  .
                  .

BIND_SUBSTRING    THE SUBSTRING
                       "{user title}"
                  TO THE OPERATION
                       OPERATION_TYPE_N
                  IN THE MAP
                       OPERATION MAP

END INITIALIZE_OPERATION_MAP
```

A.2.3.2 Execute

Execute prompts the user for a behavior identifier and an object identifier. Execute then goes to the current context's object list (unless the identifier is the reserved word "self") and gets a pointer to the object of interest. Execute then goes to the Operation_Map to get the

OPERATION_TYPE item therein bound to the behavior identifier. Execute then passes both the OPERATION_TYPE item and the OBJECT pointer to the allocated OPERATION_TASK_TYPE (see section 5.3.3.1).

```
GET THE BEHAVIOR IDENTIFIER
GET THE OBJECT   IDENTIFIER

IF    THE OBJECT IDENTIFIER  =  "self"
THEN

      THE OBJECT IS THE CURRENT CONTEXT

ELSE

      FIND THE OBJECT
           WITH THE OBJECT IDENTIFIER
           IN THE CURRENT CONTEXT'S LIST
           OF OBJECTS

      IF   OBJECT NOT FOUND
      THEN

           APOLOGIZE AND RETURN

      END IF

FIND THE BEHAVIOR
           WITH THE BEHAVIOR IDENTIFIER
           IN THE CURRENT CONTEXT'S LIST
           OF BEHAVIORS

IF   BEHAVIOR NOT FOUND
THEN

     APOLOGIZE AND RETURN

ELSE

     ALLOCATE A NEW OPERATION TASK TYPE
               TO THE OPERATION
```

```
            EXECUTE  THE OPERATION
                       THE RANGE OF THE DOMAIN
                       BEHAVIOR IDENTIFIER AS
                       BOUND IN THE OPERATION MAP
                     AGAINST THE OBJECT

          END IF
```

A.2.3.3 Utilize

Utilize is used to change the users context from the current context, to one that of one of the objects which reside in the current context's list of (sub)objects. Once the current context is changed, the user is then presented with the current Menu (see section 5.3.2.1) for that context. At this point, the user can then select a HOOKE action.

```
          GET THE OBJECT'S IDENTIFIER

          FIND THE OBJECT WITH THE OBJECT IDENTIFIER
               IN THE CURRENT CONTEXT'S LIST OF OBJECTS

          IF   OBJECT NOT FOUND
          THEN

               APOLOGIZE AND RETURN

          ELSE

               SET THE CURRENT CONTEXT TO THE OBJECT

               LOOP

                  HOOKE_Package.Menu  FOR THE CURRENT
                        CONTEXT RETURNS USER SELECTION
```

```
CASE  USER SELECTION  IS

   when  HOOKE_Package.nop       =}
         DO NOTHING

   when  HOOKE_Package.help      =}
         HELP FOR THE CURRENT CONTEXT

   when  HOOKE_Package.create    =}
         CREATE AN OBJECT
            IN THE CURRENT CONTEXT

   when  HOOKE_Package.fetch     =}
         FETCH AN OBJECT
            INTO THE CURRENT CONTEXT

   when  HOOKE_Package.modify    =}
         MODIFY AN OBJECT
            IN THE CURRENT CONTEXT

   when  HOOKE_Package.destroy   =}
         DESTROY AN OBJECT
            IN THE CURRENT CONTEXT

   when  HOOKE_Package.examine   =}
         EXAMINE AN OBJECT
            FROM THE CURRENT CONTEXT

   when  HOOKE_Package.utilize   =}
         UTILIZE AN OBJECT
            FROM THE CURRENT CONTEXT

   when  HOOKE_Package.execute   =}
         EXECUTE A BEHAVIOR
            FROM THE CURRENT CONTEXT

   when  HOOKE_Package.store     =}
         STORE AN OBJECT
            FROM THE CURRENT CONTEXT

   when  HOOKE_Package.quit      =}
         exit

END CASE

END LOOP
```

A.2.3.4 Additional procedures

User-defined procedures are required to be single parameter procedures which accept an Object_Package.OBJECT in out variable. All processing which the procedure performs is executed against the object it was passed and the results are returned in the OBJECT which the procedure passes back. Although attributes are what are actually affected by a behavior, the entire object is passed to the procedure. However, this follows directly from our abstraction that a behavior is an operation undertaken against an object by an object.

```
PROCEDURE_NAME
(
   The_Object   :  in out  Object_Package.OBJECT
)
is

   begin

      { user code }

   end  PROCEDURE_NAME;
```

A.2.4 MAIN

The MAIN procedure's pseudocode follows.

```
ALLOCATE NEW OBJECT RECORD The_Current_Context
```

```
SET The_Current_Context's IDENTIFIER TO "HOOKE"

CLEAR The_Current_Context's ATTRIBUTE LIST
CLEAR The_Current_Context's BEHAVIOR  LIST
CLEAR The_Current_Context's OBJECT    LIST

LOOP

   HOOKE_Package.Menu  FOR THE CURRENT CONTEXT
                       RETURNS USER SELECTION

   CASE  USER SELECTION  IS

      when  HOOKE_Package.nop      =}
            DO NOTHING

      when  HOOKE_Package.help     =}
            HELP FOR THE CURRENT CONTEXT

      when  HOOKE_Package.create   =}
            CREATE AN OBJECT
               IN THE CURRENT CONTEXT

      when  HOOKE_Package.fetch    =}
            FETCH AN OBJECT
               INTO THE CURRENT CONTEXT

      when  HOOKE_Package.modify   =}
            MODIFY AN OBJECT
               IN THE CURRENT CONTEXT

      when  HOOKE_Package.destroy  =}
            DESTROY AN OBJECT
               IN THE CURRENT CONTEXT

      when  HOOKE_Package.examine  =}
            EXAMINE AN OBJECT
               FROM THE CURRENT CONTEXT

      when  HOOKE_Package.utilize  =}
            UTILIZE AN OBJECT
               FROM THE CURRENT CONTEXT

      when  HOOKE_Package.execute  =}
            EXECUTE A BEHAVIOR
               FROM THE CURRENT CONTEXT
```

```
            when  HOOKE_Package.store     =}
                  STORE AN OBJECT
                     FROM THE CURRENT CONTEXT

            when  HOOKE_Package.quit      =}
                  exit

      END CASE

END LOOP

PRINT EXIT MESSAGE
```

B. Case Studies

Two actual Turing machines were implemented under the Hooke as a proof of concept for the object model. The Turing machines did execute interactively using the operation task map, however, due to the lack of the "modify" command in the Hooke, they could not be entirely developed in situ, but had to rely in part on a text editor to supply attributes and values. Please note that backslashes (\) were used in the actual code where slashes (/) appear in this appendix.

The first Turing machine is due to Hopcroft and Ullman [1979] and, in fact, is taken from example 7.1 [Hopcroft and Ullman, 1979: 149-150]. The second Turing machine is a palindrome recognizer over the alphabet {0,1,B}. These two cases can be considered instantiations of a generic Turing machine where the formal generic parameters, delta function and input tape, are replaced by the appropriate actual parameters.

B.1. Recognizer for the Language, L = { 0^n1^n | n ≥ 1}

The first case study provides a Turing machine which accepts an arbitrary number of 0's followed by the same number of 1's, as long as there is a positive number of each. Thus we define:

Q = { q0, q1, q2, q3, q4 }

S = { 0, 1 }

G = { 0, 1, X, Y, B }

F = { q4 }

And the delta function is given by table B.1.

Table B.1. Delta function for L = { 0^n1^n | n ≥ 1}

State	Symbol 0	1	X	Y	B
q0	(q1,X,R)	-	-	(q3,Y,R)	-
q1	(q1,0,R)	(q2,Y,L)	-	(q1,Y,R)	-
q2	(q2,0,L)	-	(q0,X,R)	(q2,Y,L)	-
q3	-	-	-	(q3,Y,R)	(q4,B,R)
q4	-	-	-	-	-

The actual structure of the Turing machine, delta function, and the input tapes under the HOOKE:

```
[ /TM/

  {}
  { [ /move/ {} {} ] }
  { [ /tape head/
      { [ /location/       /0/ {} ]
        [ /current input/ / / {} ]
       }
      {}
      {}
     ]
    [ /finite control/
      { [ /current state/  /q0/ {} ]
       }
      {}
      {}
     ]
   }
 ]

==============

[ /Q/

  { ATTRIBUTES }
  { BEHAVIORS }
  { OBJECTS
    [ /start/
      {}
      {}
      {
        [ /q0/ {} {} {} ]
       }
     ]

    [ /q1/ {} {} {} ]
    [ /q2/ {} {} {} ]
    [ /q3/ {} {} {} ]

    [ /final/
      {}
      {}
      {
        [ /q4/ {} {} {} ]
       }
     ]
  }
]
```

```
=============

[ /delta/

  {}  ATTRIBUTES
  {}  BEHAVIORS
  {   OBJECTS

    [ /q0/ START STATE

      {}
      {}
      {
        [ /0/

          {
            [ /next state/     /q1/   {} ]
            [ /output symbol/ /X/     {} ]
            [ /direction/      /R/    {} ]
          }
          {}
          {}
        ]

        [ /Y/

          {
            [ /next state/     /q3/   {} ]
            [ /output symbol/ /Y/     {} ]
            [ /direction/      /R/    {} ]
          }
          {}
          {}
        ]
      }
    ]


    [ /q1/

      {}
      {}
      {

        [ /0/

          {
            [ /next state/     /q1/   {} ]
```

```
        [ /output symbol/ /0/    {} ]
        [ /direction/     /R/    {} ]
      }
     {}
     {}
   ]

  [ /1/

     {
       [ /next state/    /q2/   {} ]
       [ /output symbol/ /Y/    {} ]
       [ /direction/     /L/    {} ]
      }
     {}
     {}
   ]

  [ /Y/

     {
       [ /next state/    /q1/   {} ]
       [ /output symbol/ /Y/    {} ]
       [ /direction/     /R/    {} ]
      }
     {}
     {}
   ]
  }
]


[ /q2/

  {}
  {}
  {

    [ /0/

      {
        [ /next state/    /q2/   {} ]
        [ /output symbol/ /0/    {} ]
        [ /direction/     /L/    {} ]
       }
      {}
      {}
    ]
```

```
    [ /X/

      {
        [ /next state/     /q0/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

    [ /Y/

      {
        [ /next state/    /q2/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/     /L/    {} ]
       }
      {}
      {}
     ]
   }
 ]


[ /q3/

  {}
  {}
  {

    [ /Y/

      {
        [ /next state/     /q3/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]
```

```
          [ /B/

            {
              [ /next state/     /q4/   {} ]
              [ /output symbol/ /B/     {} ]
              [ /direction/      /R/    {} ]
             }
            {}
            {}
           ]
         }
       ]

     [ /q4/ FINAL STATE

       {}
       {}
       {}
      ]

   }
 ]

=============

[ /gamma/

  {}
  {}
  {

    [ /blank/

      {}
      {}
      { [ /B/ {} {} {} ] }
     ]

    [ /sigma/

      {}
      {}
      {
        [ /0/ {} {} {} ]
        [ /1/ {} {} {} ]
       }
     ]
```

```
    [ /X/   {} {} {} ]
    [ /Y/   {} {} {} ]

  }
]

=============

[ /input tape/  --  A good tape, it should be accepted.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /3/   { [ /tape symbol/ /1/ {} ] } {} {} ]
   }
 ]

=============

[ /input tape/  --  A bad tape: illegal transition.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
   }
 ]

=============

[ /input tape/  --  A bad tape: too many 0's.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]

   }
 ]
```

```
=============

[ /input tape/  --  A bad tape:  too many 1's.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]

  }
]

=============

[ /input tape/  --  A bad tape:  input follows (0**n)(1**n)

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /3/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /4/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /5/   { [ /tape symbol/ /1/ {} ] } {} {} ]

  }
]

=============
```

The following table summarizes the results obtained by executing the TM against the above input tapes:

Table B.2. 0^n1^n Results.

input	expected results		actual results	
	final tape	accepts	final tape	accepts
0011	XXYY	Y	XXYY	Y
001	XXY	N	XXY	N
011	XY1	N	XY1	N
001101	XXYY01	N	XXYY01	N

B.2. Palindrome Recognizer.

We define palindromes over {0,1} recursively, as follows:

1) epsilon, 0, and 1 are palindromes;

2) if w is a palindrome, so are 0w0 and 1w1;

3) nothing else is a palindrome.

Thus we define:

Q = { q0, q1, q2, q3, q4, q5, q6, q7 }

S = { 0, 1 }

G = { 0, 1, X, Y, B }

F = { q7 }

And the delta function is given by table B.1 recognizes palindromes over {0,1}.

Table B.3. Delta function for palindrome recognizer.

State	Symbol 0	1	X	Y	B
q0	(q1,X,R)	(q2,Y,R)	-	-	(q7,B,R)
q1	(q1,0,R)	(q1,1,R)	(q3,X,L)	(q3,Y,L)	(q3,B,L)
q2	(q2,0,R)	(q2,1,R)	(q4,X,L)	(q4,Y,L)	(q4,B,L)
q3	(q5,X,L)	-	(q7,X,R)	-	-
q4	-	(q5,Y,L)	-	(q7,Y,R)	-
q5	(q5,0,L)	(q5,1,L)	(q6,X,R)	(q6,Y,R)	-
q6	(q1,X,R)	(q2,Y,R)	(q7,X,L)	(q7,Y,L)	-
q7	-	-	-	-	-

The actual structure of the Turing machine, delta function, and the input tapes under the HOOKE:

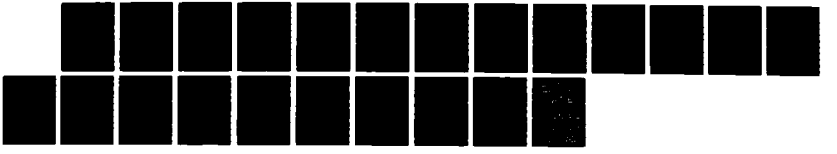
```
[ /Turing machine/

  {}
  { [ /move/ {} {} ] }
  {

    [ /tape head/
      {
        [ /location/       /0/ {} ]
        [ /current input/ / / {} ]
      }
     {}
     {}
    ]
```

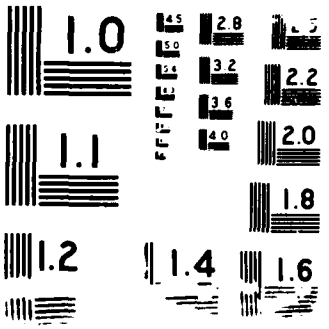
```
    [ /finite control/
      { [ /current state/  /q0/ {} ]
        }
      {}
      {}
     ]

   }
 ]

==================

[ /Q/

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /start/
      {}
      {}
      {
        [ /q0/ {} {} {} ]
       }
     ]

    [ /q1/ {} {} {} ]
    [ /q2/ {} {} {} ]
    [ /q3/ {} {} {} ]
    [ /q4/ {} {} {} ]
    [ /q5/ {} {} {} ]
    [ /q6/ {} {} {} ]

    [ /final/
      {}
      {}
      {
        [ /q7/ {} {} {} ]
       }
     ]
   }
 ]
```

```
==================

[ /delta/

   {}  ATTRIBUTES
   {}  BEHAVIORS
   {   OBJECTS

     [ /q0/ START STATE

        {}
        {}
        {

          [ /0/

            {
              [ /next state/     /q1/   {} ]
              [ /output symbol/ /X/     {} ]
              [ /direction/      /R/    {} ]
            }
           {}
           {}
          ]

          [ /1/

            {
              [ /next state/     /q2/   {} ]
              [ /output symbol/ /Y/     {} ]
              [ /direction/      /R/    {} ]
            }
           {}
           {}
          ]

          [ /B/

            {
              [ /next state/     /q7/   {} ]
              [ /output symbol/ /B/     {} ]
              [ /direction/      /R/    {} ]
            }
           {}
           {}
          ]
        }
      ]
```

AD-A194 879 AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. W A BRALICK 3/3

UNCLASSIFIED MAR 88 AFIT/GCS/MA/88M-01 F/G 12/9 NL




1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

```
[ /q1/

  {}
  {}
  {

    [ /0/
      {
        [ /next state/     /q1/   {} ]
        [ /output symbol/ /0/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

    [ /1/
      {
        [ /next state/     /q1/   {} ]
        [ /output symbol/ /1/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

    [ /B/
      {
        [ /next state/     /q3/   {} ]
        [ /output symbol/ /B/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

    [ /X/
      {
        [ /next state/     /q3/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]
```

```
    [ /Y/

      {
        [ /next state/     /q3/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

   }
 ]


[ /q2/

  {}
  {}
  {

    [ /0/

      {
        [ /next state/     /q2/   {} ]
        [ /output symbol/ /0/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

    [ /1/

      {
        [ /next state/     /q2/   {} ]
        [ /output symbol/ /1/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]
```

```
    [ /B/

      {
        [ /next state/     /q4/   {} ]
        [ /output symbol/ /B/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

    [ /X/

      {
        [ /next state/     /q4/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

    [ /Y/

      {
        [ /next state/     /q4/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

   }
 ]


[ /q3/

  {}
  {}
  {
```

```
    [ /0/

      {
        [ /next state/     /q5/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

    [ /X/

      {
        [ /next state/     /q7/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

   }
 ]


[ /q4/

  {}
  {}
  {

    [ /1/

      {
        [ /next state/     /q5/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]
```

```
    [ /Y/

      {
        [ /next state/     /q7/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

   }
 ]


[ /q5/

  {}
  {}
  {

    [ /0/

      {
        [ /next state/     /q5/   {} ]
        [ /output symbol/ /0/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]

    [ /1/

      {
        [ /next state/     /q5/   {} ]
        [ /output symbol/ /1/     {} ]
        [ /direction/      /L/    {} ]
       }
      {}
      {}
     ]
```

```
    [ /X/

      {
        [ /next state/     /q6/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

    [ /Y/

      {
        [ /next state/     /q6/   {} ]
        [ /output symbol/ /Y/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]

   }
 ]


[ /q6/

  {}
  {}
  {

    [ /0/

      {
        [ /next state/     /q1/   {} ]
        [ /output symbol/ /X/     {} ]
        [ /direction/      /R/    {} ]
       }
      {}
      {}
     ]
```

```
        [ /1/

          {
            [ /next state/     /q2/   {} ]
            [ /output symbol/ /Y/     {} ]
            [ /direction/      /R/    {} ]
           }
          {}
          {}
         ]

        [ /X/

          {
            [ /next state/     /q7/   {} ]
            [ /output symbol/ /X/     {} ]
            [ /direction/      /L/    {} ]
           }
          {}
          {}
         ]

        [ /Y/

          {
            [ /next state/     /q7/   {} ]
            [ /output symbol/ /Y/     {} ]
            [ /direction/      /L/    {} ]
           }
          {}
          {}
         ]

       }
     ]


   [ /q7/  FINAL STATE {} {} {} ]

  }
]
```

```
==================

[ /gamma/
  {}
  {}
  {

    [ /blank/
      {}
      {}
      {
        [ /B/ {} {} {} ]
      }
    ]

    [ /sigma/
      {}
      {}
      {
        [ /0/ {} {} {} ]
        [ /1/ {} {} {} ]
      }
    ]

    [ /X/  {} {} {} ]
    [ /Y/  {} {} {} ]

  }
]

==================

[ /input tape/  --  epsilon should be O.K.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /B/ {} ] } {} {} ]

  }
]
```

```
==================

[ /input tape/  --  singleton 0 should also be O.K.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]

  }
]

==================

[ /input tape/  --  Singleton 1 should be O.K., too.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]

  }
]

==================

[ /input tape/  --  Should be no good.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /1/ {} ] } {} {} ]

  }
]
```

```
==================

[ /input tape/  --  Should be O.K.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Should be O.K.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Should be O.K.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /3/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /4/   { [ /tape symbol/ /1/ {} ] } {} {} ]

   }
 ]
```

```
==================

[ /input tape/  --  Should bomb, X not in SIGMA.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /X/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Should bomb, Y not in SIGMA either.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /Y/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Oops, this isn't even defined!

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ / / {} ] } {} {} ]

   }
 ]
```

```
==================

[ /input tape/  --  Should be good.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Should reject.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /0/ {} ] } {} {} ]

   }
 ]

==================

[ /input tape/  --  Should reject.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]

   }
 ]
```

```
==================

[ /input tape/  --  Should reject.

  { ATTRIBUTES }
  { BEHAVIORS  }
  { OBJECTS

    [ /0/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /1/   { [ /tape symbol/ /0/ {} ] } {} {} ]
    [ /2/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /3/   { [ /tape symbol/ /1/ {} ] } {} {} ]
    [ /4/   { [ /tape symbol/ /1/ {} ] } {} {} ]
   }
 ]

==================
```

The following table summarizes the results obtained by executing the TM against the following input tapes:

Table B.4. Palindrome Results.

	expected results		actual results	
input	final tape	accepts	final tape	accepts
B	B	Y	B	Y
0	X	Y	X	Y
1	Y	Y	Y	Y
00	XX	Y	XX	Y
111	YYY	Y	YYY	Y
101	YXY	Y	YXY	Y
10101	YXYXY	Y	YXYXY	Y
X	X	N	X	N
Y	Y	N	Y	N
" "	" "	N	" "	N
01	X1	N	X1	N
110	Y10	N	Y10	N
001	X01	N	X01	N
10111	YX11Y	N	YX11Y	N

B.3. Conclusions.

In all cases, the Turing machine produced exactly the results predicted. The object model representation of the Turing machine is shown to be correct in these cases. Thus the Object model and its implementation under the Hooke should be able to effectively represent any "computable function" within the limits of the hardware upon which the system is implemented.

BIBLIOGRAPHY

Abbott, Russell J. "Program Design by Informal English Descriptions," Communications of the ACM, Vol. 26, No.11 (Nov 1983).

Agha, Gul. "An Overview of Actor Languages," SIGPLAN Notices, Vol 21: 58-67 (October 1986).

Berard, E. Object-Oriented Design Handbook. Rockville, MD.: EVB Software Engineering, Jan 1985.

Booch, Grady. Software Components with Ada. Menlo Park, CA.: Benjamin Cummings Publishing Co., 1987.

----- "Object-Oriented Development," IEEE Transactions on Software Engineering, SE-12: 211-221 (February 1986).

----- Software Engineering with Ada 2d ed. Menlo Park, CA.: Benjamin Cummings Publishing Co., 1986.

Brooks, F. The Mythical Man-Month. Reading, MA.: Addison-Wesley Publishing Co., 1975.

Cannon, Howard. "Flavors: A Non-heirarchical Approach to Object-Oriented Programming," Draft Report, 1982.

Cardelli, Luca and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism," Computing Surveys, 17: 471-522, (December 1985)

Cox, Brad J. Object-Oriented Programming: An Evolutionary Approach. Reading, MA.: Addison-Wesley Publishing Co., 1986.

Department of Defense. Ada Programming Language Reference Manual. ANSI-MIL-STD 1815-A-1983.

----- Requirements for Ada Programming Support Environments (Stoneman). Washington D.C.: Government Printing Office, 1980.

Dijkstra, E.W. "The Humble Programmer," Communications of the ACM, Vol. 12, No. 10. (Oct 1972).

Dittrich, Klaus and R. Lorie. Object-Oriented Database Concepts for Engineering Applications. San Jose, CA.: IBM Research Laboratory, May 8, 1985.

EVB Software Engineering Inc. An Object-Oriented Design Handbook for Ada Software. Rockville, MD., 1985.

Fairley, Richard. Software Engineering Concepts. New York: McGraw-Hill Inc., 1985.

Goldberg, Adele. SMALLTALK-80: The Interactive Programming Environment. Reading, MA.: Addison-Wesley Publishing Co., 1984.

----- and D. Robson. SMALLTALK-80: The Language and its Implementation. Reading, MA.: Addison-Wesley Publishing Co., 1983.

Guttag, J. and J. Horning. "The Algebraic Specification of Abstract Data Types," Acta Informatica, Vol. 10, No. 1, 1978.

Hopcroft, J. and J. Ullman. Introduction to Automata Theory, Languages and Computation. Reading, MA.: Addison-wesley Publishing Co.,1979.

Houghton, R. and D. Wallace. "Characteristics and Functions of Software Engineering Environments : An Overview," ACM Sigsoft Software Engineering Notes, 12: 1-43 (Jan 1987).

Institute of Electrical and Electronic Engineers. IEEE Standard Glossary of Software Engineering Terminology. New York, 1983.

Intel Corporation. iAPX 432 Object Primer. Manual 171858-001 Revision B. Aloha, OR., 1980.

Jones, Michael and R. Rashid. "Mach and Matchmaker: Kernal and Language Support for Object-Oriented Distributed Systems," OOPSLA '86 Proceedings, Portland, OR., September 29-October 2, 1986: 67-77 (Sep 1986).

Kreutzer, W. System Simulation Programming Styles and Languages. Sydney: Addison-Wesley Publishing Co., 1986.

Lieberman, Henry. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," OOPSLA '86 Proceedings, 214-223 (Sep 1986).

Liskov, Barbara, et. al. "Abstraction Mechanisms in CLU," Communications of the ACM, Vol. 20, No. 8: 564-576 (Aug 1977).

----- and S. Zilles. "An Introduction to Formal Specifications of Data Abstractions," Current Trends in Programming Methodology. Englewood Cliffs, NJ.: Prentice Hall, 1977.

MacLennan, B.J. A View of Object-Oriented Programming. Naval Postgraduate School, NPS52-83-001, Feb 1983.

Meyrowitz, Norman. "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," OOPSLA '86 Proceedings, 186-201 (Sep 1986).

Nygaard, Kristen. "Basic Concepts in Object Oriented Programming," SIGPLAN Notices, 21: 128-132 (October, 1986).

Organick, E. A Programmer's View of the Intel 432 System. New York: McGraw-Hill , 1983.

Oxford English Dictionary, The, Volume 3. Oxford: Clarendon Press, 1933.

Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM (Dec 1972).

Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," Byte, 11: 139-144 (August 1986).

Pratt, Terrence. Programming Languages: Design and Implementation. Englewood Cliffs, NJ.: Prentice Hall Inc., 1975.

Pressman, Roger. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 1987.

Random House College Dictionary. New York: Random House Inc., 1984 edition.

Schmucker, Kurt J. "Object-Oriented Languages for the MacIntosh," Byte, 11: 177-185 (August 1986).

Shaw, Mary. "Abstraction Techniques in Modern Programming Languages," IEEE Software, Vol. 1, No. 4 (Oct 1984).

Simon, Herbert. Sciences of the Artificial, The. Cambridge, MA.: MIT Press, 1985.

Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages," OOPSLA '86 Proceedings: 38-45 (September 1986).

Steele, Guy. Common Lisp-The Language. Burlington, MA.: Digital Press, 1984.

Ullman, J.D. Principles of Database Systems, 2nd edition. Rockville, MD.: Computer Science Press, 1982.

Williams, Gregg. "Hypercard," Byte, Vol. 12, No. 14 (Dec 1987).

VITA

Captain William A. Bralick, Jr. was born on April 29, 1954 in Watertown, Wisconsin. He graduated from Menomonee Falls North High School in June of 1972. He enlisted in the Air Force in May of 1973, and served four years as a Russian Linguist. Following his enlisted service, Captain Bralick attended the University of Wisconsin at Madison, from which he obtained a Bachelor of Science degree in Computer Science in 1980.

He received a commission the following year, upon graduation from the Officer Training School at Lackland Air Force Base. He has since served as a computer systems engineer at the 6944th Electronic Security Squadron at Fort Meade, Maryland, and with the IFFN Joint Test Force at Kirtland AFB, New Mexico. He entered the Air Force Institute of Technology, School of Engineering, in June of 1986.

Permanent Address: c/o Bertrand Dyer
RR 1 Box 240
Kingfield
ME 04947

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; Distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/MA/88M-01	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: SCHOOL OF ENGINEERING; 6b. OFFICE SYMBOL (If applicable): AFIT/ENC	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): AIR FORCE INSTITUTE OF TECHNOLOGY, WRIGHT PATTERSON AFB, OH 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
AN EXAMINATION OF THE THEORETICAL FOUNDATIONS OF THE OBJECT-ORIENTED PARADIGM (U)

12. PERSONAL AUTHOR(S)
BRALICK, WILLIAM A.,JR., CAPT., USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS THESIS	FROM _____ TO _____	1988 March	220

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	OBJECT, OBJECT-ORIENTED, COMPUTABILITY
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

THESIS CHAIRMAN: DAVID A. UMPHRESS, CAPT., USAF
ASSISTANT PROFESSOR OF MATHEMATICS AND COMPUTER SCIENCE

ABSTRACT: (see reverse)

Approved for public release: IAW AFR 190-1.
14 Mar 88
Dean for Research and Professional Development

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: CAPT. DAVID A. UMPHRESS	22b. TELEPHONE (Include Area Code): 513-255-3098; 22c. OFFICE SYMBOL: AFIT/ENC

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

The object-oriented paradigm provides a natural structure for describing and decomposing systems. The objectives of this research were to (1) provide a definition of an object model and consider its theoretical foundations, (2) implement the defined object model to empirically investigate the concept, and (3) implement a prototype environment to directly, interactively manipulate the object model.

We define an object to have a unique identity and be composed of a set of attributes, a set of behaviors, and a set of (sub)objects. We define an attribute to be composed of an identifier, a value, and a set of attributes; and we define a behavior to be an identifier, a set of attributes, and a set of behaviors. We propose and prove the theorem that the defined object model can simulate a Turing machine.

We then use the object-oriented design process to implement the defined object model under a prototype intractive environment called the HOOKE. We use the HOOKE to help build a simulation of a Turing machine under the defined object model, including several delta functions and input tapes. Thus we validate both the defined object model and the HOOKE. We conclude that the object-oriented paradigm rests on sound theoretical ground.

END

DATE

FILMED

8-88

DTIC